



COMPONENT-ORIENTED PROGRAMMING

ANDY JU AN WANG
KAI QIAN



COMPONENT-ORIENTED PROGRAMMING

ANDY JU AN WANG

KAI QIAN

Southern Polytechnic State University
Marietta, Georgia



A JOHN WILEY & SONS, INC., PUBLICATION

COMPONENT-ORIENTED PROGRAMMING

COMPONENT-ORIENTED PROGRAMMING

ANDY JU AN WANG

KAI QIAN

Southern Polytechnic State University
Marietta, Georgia



A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2005 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400, fax 978-646-8600, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

Library of Congress Cataloging-in-Publication Data:

Wang, Andy Ju An.

Component-oriented programming / Andy Ju An Wang & Kai Qian.

p. cm.

“A Wiley-Interscience publication.”

Includes bibliographical references.

ISBN 0-471-64446-3

1. Component software. 2. Computer programming. I. Qian, Kai. II. Title.

QA76.76.C66W36 2005

005.3 – dc22

2004015681

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CONTENTS

Preface	ix
1 Introduction	1
1.1 What Is COP? 1	
1.2 Why Is COP Important? 3	
1.3 What Is a Component? 5	
1.4 Hardware Components and Software Components, 7	
1.5 From OOP to COP, 8	
1.6 Component-Based Software Engineering, 10	
1.7 Summary, 11	
1.8 Self-Review Questions, 12	
1.9 Exercises, 14	
References, 14	
2 Theory of Components	16
2.1 Principles of COP, 16	
2.2 Infrastructures of COP, 18	
2.3 Component Models, 20	
2.4 Connection Models, 21	
2.5 Deployment Models, 26	
2.6 Unifying Component Technologies, 26	
2.7 Summary, 32	
2.8 Self-Review Questions, 33	
2.9 Exercises, 35	
References, 35	
3 COP with JavaBeans	37
3.1 Overview of JavaBeans Technology, 37	
3.2 Component Model of JavaBeans, 38	

3.3	Connection Model of JavaBeans,	64
3.4	Deployment Model of JavaBeans,	72
3.5	Examples and Lab Practice,	76
3.6	Summary,	83
3.7	Self-Review Questions,	85
3.8	Exercises,	86
	References,	87
4	Enterprise JavaBeans Components	88
4.1	EJB Architecture,	88
4.2	Component Model of EJB,	90
4.3	Connection Model of EJB,	100
4.4	Deployment Model of EJB,	104
4.5	Examples and Lab Practice,	105
4.6	Summary,	142
4.7	Self-Review Questions,	143
4.8	Exercises,	143
4.9	Programming Exercises,	144
	References,	145
5	CORBA Components	146
5.1	CORBA Component Infrastructure,	146
5.2	CORBA Component Model (CCM),	149
5.3	Connection Model of CORBA and CCM,	173
5.4	Deployment Model of CORBA and CCM,	176
5.5	Examples and Lab Practice,	184
5.6	Summary,	189
5.7	Self-Review Questions,	190
5.8	Exercises,	191
5.9	Programming Exercises,	192
	References,	192
6	.NET Components	194
6.1	.NET Framework,	194
6.2	Component Model of .NET,	198
6.3	Connection Model of .NET,	204
6.4	.NET Component Deployments,	212
6.5	Visual Studio .NET,	215
6.6	Examples and Lab Practice,	224
6.7	Summary,	234
6.8	Self-Review Questions,	234
6.9	Exercises,	235
6.10	Programming Exercises,	236
	References,	237
7	COP with OSGi Components	238
7.1	Overview of OSGi Technology,	238
7.2	Component Model of OSGi,	239

7.3	Connection Model of OSGi,	247
7.4	Deployment Model of OSGi,	250
7.5	Examples and Lab Practice,	252
7.6	Summary,	261
7.7	Self-Review Questions,	263
7.8	Exercises,	265
	References,	266
8	Web Services Components	267
8.1	Web Services Framework,	267
8.2	Component Model of Web Services,	281
8.3	Connection Model of Web Services,	293
8.4	Web Services Component Deployment,	297
8.5	Examples and Lab Practice,	299
8.6	Summary,	311
8.7	Self-Review Questions,	312
8.8	Exercises,	313
8.9	Programming Exercises,	314
	References,	314
Index		315

PREFACE

This book is about component-oriented programming (COP for short), a new programming paradigm beyond object-oriented programming. Component-oriented programming offers higher reusability and better modular structure with greater flexibility, compared with object-oriented programming or library-based programming approaches. This book discusses principles of component-oriented programming and provides a unified component infrastructure to describe different component technologies. Hands-on programming practice is emphasized in this book. Readers of this book will learn how to develop reusable software components and how to build a software system out of prebuilt software components.

Both authors have been teaching “Component-Based Software Development” (CBSD) to graduate and undergraduate students for several years. We strongly believe that we need a textbook emphasizing the “programming” aspect of CBSD or CBSE (Component-Based Software Engineering). Thus, students majoring in computer science, software engineering, or related areas can take this course as one of the major “concentrations” following OOP (Object-Oriented Programming) and elementary software engineering courses. CBSD is not just a philosophy, a theory, or some standard. It represents a new programming paradigm beyond OOP and library-based programming paradigms – this is partially why we chose the title *Component Oriented Programming*. In a library-based programming paradigm, multiple layers of software abstractions stack up one on top of the other, whereas in a component-based model, multiple software components plug in side-by-side with one another. In a library-based model, once a module is compiled into the software, it is always available at runtime. In a component-based model, a component can come and go dynamically at runtime. In a library-based paradigm, one most likely must rebuild and repackage the entire set of libraries to fix bugs and add features at compile-time. In a component-based paradigm, new components can be added or existing components can be updated in an incremental way at runtime. In a library-based paradigm, changes made to public interfaces have less impact because one has to rebuild the software in its entirety. In a component-based paradigm, the cost of

redoing public interfaces can be prohibitive because other components may have relied on them at runtime.

We can make a long list of differences between traditional, library-based programming and new, component-based programming. Many software engineers have not realized or appreciated this shift in programming paradigms. As many experts have predicted, CBSE and component-oriented programming will be the future of software development following structured programming, library-based programming, and object-oriented programming. It is essential to make students realize that component-based software development requires more discipline on the part of developers during the design and programming stages to provide more stability, extensibility, and flexibility. Such disciplined activity and techniques have to be established through well-developed training and practice following component-based software development principles.

However, books published on CBSE are not suited to provide such training. Some are highly theoretical, targeting academic researchers, while others emphasize software life cycle and project management; yet others are too specific for particular component technology. In a typical computer science or software engineering curriculum setting in colleges and universities, the course for “component-based software development” is usually designed for students with previous training on OOP and fundamental software engineering principles. Hence, it is challenging to shift students from the mindset of library-based programming and OOP to the mindset of component-oriented programming. Both authors have followed a “programming-centered approach” to teaching the “Component-Based Software Development” course, which has turned out to be successful in emphasizing programming methodology and programming practice in the CBSE context.

The programming paradigm is changing from object-oriented programming to component-oriented programming. We predict that in the next few years, component-based development will be the mainstream practice in software engineering. This book will provide in-depth knowledge about component-based development. It intends to introduce component-oriented programming to college students majoring in computer science, software engineering, information technology and to professional software engineers working in the industry as well. We assume that the readers of the book are familiar with at least one high-level programming language. The emphasis will be on component-oriented programming principles, methods, and techniques. Any reader who is interested in component-based software engineering in general will find useful information in this book. Readers can expect to gain the comprehensive knowledge necessary for component software development.

Three efforts have been made to help readers understand and master component-oriented programming techniques. First, a unified component infrastructure is used for all component technologies covered in this book, namely, JavaBeans, EJB, COM/DCOM/.NET, CORBA, OSGi, and Web Services. This common component infrastructure contains a component model, a connection model, and a deployment model. It represents standard conventions for component specification, interfaces, composing connectors, architecture, packaging, and deployment. It also helps to apply the component principles to practical component software development following a well-defined format. The unified component infrastructure also serves as a pedagogical tool to keep readers concentrating on technical essentials rather than being distracted by those technical details.

Second, we have paid careful attention to the pedagogical organization of the book. For instance, we put forward briefly at the beginning of each chapter its main objectives, and we explain all potential tricky points in detail in the process of presenting the content. At the end, a brief summary is provided to highlight the key points of the chapter. *Self-Review Questions* are designed for readers to test their understanding of the materials in each chapter, and key solutions for these questions are supplied. *Exercises and Lab Practice* are provided at the end of each chapter, serving as an integral part of the book.

Third, a supplementary Web site (<ftp://ftp.wiley.com/public/sci tech med/component/>) is provided containing all examples, laboratory project guidelines, software packages, and developing tools used in the book.

Overview of Chapters

Chapter 1 is a brief introduction to component-oriented programming and the benefits of component-based software development.

Chapter 2 discusses the principles of component-oriented programming, and provides a unified framework for describing different component technologies.

Chapter 3 introduces the component infrastructure of JavaBeans. Both BDK (Bean Development Kit) and Bean Builder are discussed.

Chapter 4 covers J2EE framework and EJB component architecture.

Chapter 5 discusses CORBA (Common Object Request Broker Architecture) component infrastructure.

Chapter 6 introduces Microsoft .NET framework and Visual Studio .NET component technology.

Chapter 7 covers the component technology proposed by OSGi (Open Service Gateway initiative).

Chapter 8 investigates Web services component technology.

Acknowledgments

Thanks to Cassie Craig of John Wiley & Sons, Inc., for her incredibly positive support in the writing of this text. We want to thank all the other people who helped improve the text, from the reviewers and editors to those who sent suggestions and feedback in the early drafts of this book.

1

INTRODUCTION

Objectives of This Chapter

- Define COP and its major objectives
- Discuss the importance of COP
- Define components and their characteristics
- Distinguish hardware components and software components
- Investigate the differences between OOP and COP
- Introduce component-based software engineering

1.1 WHAT IS COP?

Throughout this book COP stands for *component-oriented programming*.

Programming is an activity of constructing computer programs that are a sequence of instructions describing how to perform certain tasks with computers. Programming can be classified or described in terms of the major techniques, concepts, or facility used for constructing computer programs. Different computers require different kinds of programming techniques. The Pascal's machine built in 1642 by Blaise Pascal (1623–1662), for instance, could only be programmed by operating gears and cranks. For all modern electronic computers, however, certain programming languages have to be used to program the computers. If a specific technique, say *X*, is being used to program computers, we will say this is *X-oriented programming*. Below is a brief discussion of different programming paradigms toward component-oriented programming.

Gear-Oriented Programming

Similar to the Pascal's machine mentioned above, for Charles Babbage's difference engine in 1822, programming means to change gears, cogs, and wheels. Through these physical motion and arrangement of gears, a new computation could be performed. Obviously, this kind of computer programming required both mental intelligence and physical capability as well.

Switch-Oriented Programming

For the first electronic computer ENIAC (Electronic Numerical Integrator And Computer) in 1942, programming means presetting switches and rewiring the entire system for each new "program" or calculation. This could be a tedious work because ENIAC had 6000 multiposition switches connecting a multitude of sockets with a veritable forest of jumper cables.

Procedure-Oriented Programming

With the help of high-level programming languages, programmers have been liberated from manipulating hardware details and the chores of machine languages. Programming languages such as Fortran, Pascal, and C encourage programmers to think in terms of procedures or functions. This was highlighted by structured programming, a top-down design and stepwise refinement design approach. In order to develop a computer program, the programmer would first think about what functionality should be implemented. Each functional requirement will be implemented eventually as a collection of procedures or functions.

Object-Oriented Programming

Object-oriented programming (OOP) encourages programmers to think in terms of data types. Programs are constructed around data types and data-type hierarchies. The fundamental building blocks in an object-oriented program are classes and objects. An object is a packet of information stored in a chunk of computer memory. Every object is associated with a data type, and the data type determines what can be done to an object. All programming languages have built-in data types, such as the integer data type and the character data type. Typically, programmers define data types and data-type hierarchies themselves so that they can describe individual entities to come up naturally in their applications.

Aspect-Oriented Programming

Aspect-oriented programming (AOP) introduces a new technology for separation of concerns in software development. AOP makes it possible to modularize crosscutting aspects of a system. Like objects, aspects may arise at any stage of the software life cycle, including requirements specification, design, implementation, and so on. Common examples of crosscutting aspects are design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features. AOP blends support for many different kinds of modularity, including block structure, object structure, inheritance as well as crosscutting.

A key intuition underlying AOP is that simple hierarchies are not rich enough to capture complex structures. Therefore, AOP tries to explore a variety of mechanisms that make it possible to view and implement a system from multiple perspectives. Toward this end, a new programming language called *AspectJ* was designed and implemented. *AspectJ* is a seamless aspect-oriented extension to the Java programming language, enabling the clean modularization of crosscutting concerns such as error checking

and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multiobject protocols.

Component-Oriented Programming

Component-oriented programming enables programs to be constructed from prebuilt software components, which are reusable, self-contained blocks of computer code. These components have to follow certain predefined standards including interface, connections, versioning, and deployment.

Components come in all shapes and sizes, ranging from small application components that can be traded online through component brokerages to the so-called large grain components that contain extensive functionality and include a company's business logic. In principle, every component is reusable independent of context, that is to say, it should be ready to use wherever from wherever.

COP is to develop software by assembling components. While OOP emphasizes classes and objects, COP emphasizes interfaces and composition. In this sense, we could say that COP is an interface-based programming. Clients in COP do not need any knowledge of how a component implements its interfaces. As long as interfaces remain unchanged, clients are not affected by changes in interface implementations.

1.2 WHY IS COP IMPORTANT?

How do other more mature industries or engineering disciplines deal with the development of complex systems?

Building systems out of components is a natural part of engineering systems. The automotive industry, for instance, develops very complex cars using components of every size from a tiny screw to complex subsystems such as engines and transmissions. The modern automotive factory has become more of a system integrator than a manufacturer. It is easy to name many other industries and engineering disciplines making effective use of components by a rigorous set of standards that define interoperability. For hundreds of years, industries have adopted component standards for interchangeable parts and streamlined assembly tools to speed the development of highly complex products.

It is the Industry Revolution that dramatically changed the nature of production in which machines replaced tools, steam and other energy sources replaced human or animal power, unskilled workers replaced skilled workers, and massive products made by machines and assemble lines replaced handmade items.

In the software industry, however, the products are still mainly handmade items. The productivity is low, the quality is not guaranteed, and projects are mostly overrun. The phenomenon has been called "software crisis."

As hardware technology advances, the cost of developing a computer application is mainly on software part. The key issues of software engineering include how to create quality software effectively. Components are widely seen by software engineers as an important technology to address the "software crisis." The Industrial Revolution of software will occur through component-based software engineering.

There are a number of important reasons why COP is important. It provides a higher level of abstraction. There are an increasingly large number of reusable component libraries that assist in the development of applications for various domains.

There are three major goals of COP: conquering complexity, managing change, and reuse.

1. *Conquering Complexity* We are living in a complex world at information explosion age. According to a research team at UC Berkeley [SIMS 2000], “The world produces between 1 and 2 exabytes of unique information per year, which is roughly 250 megabytes for every man, woman, and child on earth. An exabyte is a billion gigabytes, or 10^{18} bytes.” In computer science, the size and complexity of software have been increasing significantly. Table 1.1 shows the comparison of some operating systems in terms of their sizes. The first string in each box is the version; the second is the size measured in lines of source code, where $K = 1000$ and $M = 1,000,000$. Comparisons within a column have real meaning, while comparisons across columns do not [Tanenbaum 2001]. Fortunately, COP provides an effective way to deal with the complexity of software: divide and conquer.
2. *Managing Change* Change is inherent in software engineering. The user requirements change, specifications change, personnel change, budget changes, technology changes, and so on. One of the fundamental software engineering principles is to emphasize the importance of managing change. It is important to place primary

TABLE 1.1. Software Size Changes Over Time

Year	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9K					
1979	V7 21K					
1980		4.1 38K				
1982	Sys III 58K					
1984		4.2 98K				
1986		4.3 179K				
1987	SVR3 92K		1.0 13K			
1989	SVR4 280K					
1991				0.01 10K		
1993		Free 1.0 235K			5.3 850K	3.1 6M
1994		4.4 Lite 743K		1.0 165K		3.5 10M
1996				2.0 470K		4.0 16M
1997			2.0 62K		5.6 1.4M	
1999				2.2 1M		
2000		Free 4.0 1.4M			5.8 2.0M	2000 29M

emphasis during architecture and design on the dependencies between the components, and on the management of those dependencies.

COP provides an effective way to follow the software engineering principle of dealing with change: planning for change, design for change, and building for change. Components are easy to adapt to new and changing requirements. Software engineers have come to the consensus that the best way of dealing with constant changes is to build systems out of reusable components conforming to a component standard and plug-in architecture.

3. *Reuse* Software reuse allows to design and implement something once and to use it over and over again in different contexts. This will realize large productivity gains, taking advantage of best-in-class solutions, the consequent improved quality, and so forth.

There are different levels of software reuse. Source code copy, for instance, is the lowest level of reuse. Procedural function libraries are a better form of reuse than source code copy, but not extensible. Class libraries are a better form of reuse, and they are extensible. However, it requires a lot of understanding before classes can be reused. Moreover, it supports only white-box reuse; clients will be affected if internals of classes changed. For instance, in an OOP language such as C++ or Java, derived classes are coupled to the base class implementation. Changes in any of the base classes in the inheritance hierarchy would break derived classes. Furthermore, this level of reuse is language specific; no reuse across code in other languages.

COP supports highest level of software reuse because it allows various kinds of reuse including white-box reuse, gray-box reuse, and black-box reuse. White-box reuse means that the source of a software component is made available and can be studied, reused, adapted, or modified. Black-box reuse is based on the principle of information hiding. The interface specifies the services a client may request from a component. The component provides the implementation of the interface that the clients rely on. As long as the interfaces remain unchanged, components can be changed internally without affecting clients. Gray-box reuse is somewhere in between white-box reuse and black-box reuse.

As the size and complexity of software systems grows, the identification and proper management of interconnections among the pieces of the system becomes a central concern. COP provides a manageable solution to deal with the complexity of software, the constant change of systems, and the problems of software reuse. COP is now the de facto paradigm for developing large software systems, for example, enterprise-scale distributed applications, N-tier Web applications, and Web services.

1.3 WHAT IS A COMPONENT?

Software engineering has made great progress during the last 30 years with many innovative technologies invented and applied including structured programming, CASE technology, and object-oriented technology. However, the essential style of developing software maintains the same: the programmers write code line by line until it finishes. Such a preindustry style of production is the key factor for the unmatch between hardware productivity and software productivity. The revolutionized changes in software

6 INTRODUCTION

development, like the industrial revolution to classical engineering disciplines, have to be introduced:

- The handcraft fashion of software development should be replaced by engineering methods.
- The line-by-line-coding style should be replaced by component-based development.
- The syntax-based programming should be replaced by assembling components based on their interface and semantics.

Obviously, such a revolutionary change would not occur until we could discover systematic, formal approaches to component-based software development.

Software components are defined in various ways from similar and different points of view. [Brown 1998] presented four definitions of a software component, summarizing the First International Workshop on CBSE in April 1998:

1. A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. (*Philippe Krutchen, Rational Software*)
2. A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. (*Gartner Group*)
3. 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 third-party composition. (*Clemens Szyperski*)
4. A business component represents the software implementation of an “autonomous” business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. (*Wojtek Kozaczynski, SSA*)

Discussion about component definitions can also be found in many recent conferences and publications [Koza 1999; Parrish 1999; Wang 2000; Yacoub 1999; Fischer 2002; Fukazawa 2002]. In this book, we will use the following definition for a software component:

A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface.

From this definition, a component is a program or a collection of programs that can be compiled and made executable. It is self-contained; thus, it provides coherent functionality. It is self-deployable so that it can be installed and executed in an end user’s environment. It can be assembled with other components so that it can be reused as a unit in various contexts. The integration is through a component’s interface, which means that the internal implementation of a component is usually hidden from the user.

Component technologies complying with the above definition include JavaBeans and enterprise Java Beans (EJB) from Sun Microsystems, COM (Component Object

Model), DCOM (Distributed Component Object Model), and .NET components from Microsoft Corporation, and CORBA (Common Object Request Broker Architecture) components from the Object Management Group. We will discuss all these components along with OSGi (Open Service Gateway Initiative) component model and Web service components in this book.

1.4 HARDWARE COMPONENTS AND SOFTWARE COMPONENTS

For the last 50 years, the technology for producing computer hardware has radically changed. Chips of integrated circuits (IC) and very large scale integrated circuits (VLSI) make the basic building blocks from which faster and less expensive computers can be assembled. In hardware engineering, component-based approach is widely utilized to build new products through prebuilt hardware components. There are some driving factors for component-based hardware design:

- The complexity and capability of hardware devices are increasing exponentially.
- The productivity and time-to-market requirements are getting more stringent.
- Synthesis from very high-level descriptions does not provide adequate quality of results.

Hardware engineers achieve required design productivity by assembling reusable blocks of intellectual property (IP) such as microprocessors, DSP, encryption/decryption chips, and so on. Recently, silicon capability has enabled integration of entire systems on a single die. The component-based approach has increased productivity and reliability of end products since each chip component has been well tested and fabricated massively. However, there was no dramatic change in software production. Each new software product required that software designers and programmers start from scratch and produce program code line by line until the program was finished. The big progress of software development for the last 50 years is from producing a program line by line using machine code to producing a program line by line using high-level programming languages. High-level programming languages offer several important benefits that improve programming productivity thanks to compiler techniques. Nevertheless, the line-by-line fashion of software production is still dominating the current practice and becoming the bottleneck of rapid developing quality software with low cost. What are the major barriers of applying component-based approach to software engineering? What are the differences between hardware components and software components?

Logically, computer hardware and software are equivalent in terms of their computability. That is, any operation performed by software can also be built directly into the hardware. On the other hand, any instruction executed by the hardware can also be simulated in software. Typical hardware components are ICs including medium-scale integration (MSI), large-scale integration (LSI), and VLSI circuits. These chip-level components are possible because of logic circuits that have well-defined input–output functions that do not require interaction with outside world directly. The communications among hardware components are supported by a system bus. A software component, on the other hand, is usually difficult to be expressed as simply a function from an input domain to an output codomain. Moreover, some software components

must interact with users or other physical plant in the environment; thus, they cannot be treated simply as black boxes like their hardware counterparts.

It is important to note that there is a well-defined computing model, that is, Boolean algebra, for gate-level design in building hardware components. Boolean algebra provides an economical way of describing the function of digital circuitry. Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function through algebraic laws.

What is the computing model for component-based software engineering? Obviously, a similar computing model is greatly in need for component-based software development, like Boolean algebra for component-based hardware engineering. As a matter of fact, Hoare logic [Hoare 1969], among others, provides a formalism, verifying the correctness of programs based on the program structure and their pre- and postconditions. As we move from statement-level programming or line-by-line coding to component-based development, we are in need of a formalism capturing the behavior of components and assuring the correctness of component interactions. We will discuss this issue extensively in Chapter 2.

Furthermore, we have to note that the comparison of software components to hardware components is limited for two reasons. The first is that unlike memory chips – which must be physically reproduced each time, that is, producing components with slight variations – producing software components is an exact duplicate each time. The second is that a software component’s environment is likely to change for each new installation.

Finally, there is no temporal aspect, such as wear or chemical deterioration, to consider in software components [Mason 2002]. This difference is important especially when we consider the quality assurance issues for component-based development. For instance, the reliability for a hardware component is a function of four factors:

- errors in design
- errors in manufacture
- physical defects
- chemical and physical wear

The reliability for a software component, however, is a function of only one factor: errors in design (for a software component, here we consider its implementation as a part of design), because software does not wear out, nor, in most senses, are there manufacturing errors.

1.5 FROM OOP TO COP

1.5.1 Language View of COP

A program is an assembly of basic programming elements. In a machine language, the basic programming elements are 0s and 1s. Assembly languages use simple “words” as basic programming elements to abstract binary strings. Higher-level languages have made coding closer to human readable. The basic programming elements in a typical higher-level programming language are functions, also called *procedures* or *subroutines*. In an OOP (object-oriented programming) language, an object is a basic

programming element encapsulating data and behavior. Therefore, OOP is a way of partitioning functionality in modeling a problem or process into its constituent objects. The data abstraction provides a way of organizing data with associated operations, allowing classes or objects to be reused easily in another program. In COP, the basic programming elements are components, self-contained and self-deployable entities encouraging higher level of software reuse.

1.5.2 Software Engineering View of COP

OOP was claimed to support encapsulation, inheritance, and polymorphism. However, it never reached its goal because implementation inheritance violated true encapsulation. Furthermore, objects or classes are not self-deployable. We could distinguish COP from OOP in terms of various software engineering aspects as below:

- COP is interface-based, while OOP is object-based.
- COP is a packaging and distribution technology, while OOP is an implementation technology.
- COP supports high-level reuse, while OOP supports low-level reuse.
- COP, in principle, can be written in any language, while OOP is bound to OO languages.
- COP has loosely coupled components, while OOP has tightly coupled objects dependent on each other through inheritance implementation.
- COP has large granularity components, while OOP has objects as fine-grained units of composition.
- COP supports multiple interfaces, and interface-oriented design, while OOP does not provide clear relationship of interfaces among superclasses and subclasses.
- COP supports more forms of dynamic binding and dynamic discovery, while OOP provides limited support for object retrieval and runtime composition mechanisms.
- COP has better mechanisms for third-party composition, while OOP has limited forms of connectors (method invocation).
- COP provides more support for higher-order services (security, transactions, etc.), while OOP has limited sets of supported services such as security, transactions, and so on.
- COP components are designed to obey rules of the underlying component framework, while OOP objects are designed to obey OO principles.

Table 1.2 gives a brief summary of commonality and differences among structured programming (SP), OOP, and COP.

In terms of composing capability, SP is the lowest, OOP is high, and COP is very high. Two different implementation units in SP can never be interchangeable. In OOP, however, two different objects implementing the same specification are interchangeable. Yet, in COP, two different components with different specifications are interchangeable, as long as they satisfy those interface requirements for all client components. For instance, one server component may replace another one even if it has a different specification, as long as its specification includes the same interfaces the client components require.

TABLE 1.2. Comparison of COP with OOP and SP

Capabilities	SP	OOP	COP
<i>Divide and Conquer</i> • Manage complexity • Break a large problem down into smaller pieces	Yes	Yes	Yes
<i>Unification of Data and Function</i> • A software entity combines data and the functions processing those data • Improves cohesion		Yes	Yes
<i>Encapsulation</i> • The client of a software entity is insulated from how that software entity's data is stored or how its functions are implemented • Reduces coupling		Yes	Yes
<i>Identity</i> • Each software entity has a unique identity		Yes	Yes
<i>Interface</i> • Represents specification dependency • Divides a component specification into interfaces • Restricts inter-component dependency			Yes
<i>Deployment</i> • The abstraction unit can be deployed independently			Yes

1.6 COMPONENT-BASED SOFTWARE ENGINEERING

Sometimes, COP and CBSE (component-based software engineering) are interchangeable in literature. However, CBSE is a more general term, taking COP as only a part of it:

$$\text{CBSE} = \text{COA} + \text{COD} + \text{COP} + \text{COM},$$

where COA, COD, and COM represent component-oriented analysis, component-oriented design, and component-oriented management respectively. CBSE promises to accelerate software development and to reduce costs by assembling systems from pre-fabricated software components. Designing, developing, and maintaining components for reuse is, however, a very complex process, which places high requirements not only for the component functionality and flexibility but also for the development organization. CBSE covers many software engineering disciplines and different techniques, which still have not been fully defined, explained, and exploited either from theoretical or practical points of view.

In traditional software engineering, software development process consists of a sequence of activities or stages, namely, analysis, design, programming, testing, and integration. In CBSE, the main development stages become analysis, design, provision, and assembling. That is to say, the traditional programming, testing, and integration activities are replaced in CBSE by component provision and component assembling.

From the viewpoint of components, there are two kinds of activities in CBSE: *developing for reuse* (DF) and *developing with reuse* (DW). For DF, the development effort could be organized following traditional software engineering approaches, with the emphasis of component standards. For instance, each component provides two kinds of interfaces: (1) *provided interface*, which defined the public services this component will provide and (2) *required interface*, which specifies the services this component requires in order to work properly. For DW, software component search and retrieval have now become crucial activities for building applications.

From the viewpoint of engineering process, components could be classified into five different forms [Cheesman 2001]:

1. *Component Specification* This form represents the specification of a unit of software that describes the behavior of a set of *component objects* and defines a unit of implementation. Behavior is defined as a set of *interfaces*. A component specification is realized as a *component implementation*.
2. *Component Interface* The *interface* form presents a definition of a set of behaviors that can be offered by a *component object*.
3. *Component Implementation* The *implementation* form is a realization of *component specification*, which is independently deployable. This means it can be installed and replaced independently of other components. It does not mean that it is independent of other components – it may have many dependencies. It does not necessarily mean that it is a single physical item, such as a single file.
4. *Installed Component* The *installed* form is an installed (or deployed) copy of a *component implementation*. A component implementation is deployed by registering it with the runtime environment. This enables the runtime environment to identify the *installed component* to use when creating an instance of the component or when running one of its operations.
5. *Component Object* A *component object* is an instance of an *installed component*. This is a runtime concept. Similar to OOP, a component object in COP is an object with its own data and a unique identity, which performs the implemented behavior. An *installed component* may have multiple *component objects* (which require explicit identification) or a single one (which may be implicit).

1.7 SUMMARY

- With the advances of technology in computer hardware and software, the field of computer programming has grown from machine-code programming, via object-oriented programming, to component-oriented programming.
- Software components support black-box reuse.

- Software components encapsulate functionality and provide services through well-defined interfaces.
- Components are interchangeable software parts.
- A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface.
- Logically, hardware components and software components are equivalent, but COP needs fundamental theory and supporting tools to match the advance of hardware engineering.
- Compared with structured programming and OOP, the major features of COP include interfaces and independent deployment.
- There are two kinds of activities in component-based software engineering: development for reuse and development with reuse.
- From the viewpoint of software engineering, a component can take different forms: specification, interface, implementation, installed, and running object.

1.8 SELF-REVIEW QUESTIONS

- 1.** The essential characteristics of a software component include
 - a. composable
 - b. interface
 - c. deployable
 - d. all of the above
- 2.** Which of the following is common in hardware and software components?
 - a. Binary form
 - b. Unique return value
 - c. Reusable
 - d. Having Boolean logic as computing model
- 3.** Which of the following could be a component?
 - a. A macro in C
 - b. A template in C++
 - c. A block in Smalltalk
 - d. A device bean
- 4.** The major problem of Clemens' definition for a component is:
 - a. Composability should not be a characteristic of components.
 - b. Independence should not be a characteristic of components.
 - c. Components are not in binary form.
 - d. It does not reflect various forms of components.
- 5.** Which of the following is false?
 - a. Type definitions and Ada generics both can be components.
 - b. Procedures, classes, modules, or even entire applications could be components.

- c. Insisting on independence and binary form for components could allow for multiple independent vendors and robust integration.
 - d. Composition of components enables prefabricated things to be reused by rearranging them in ever-new composites.
6. CBSE is about
- a. how to create reusable components
 - b. how to create software products with reusable components
 - c. how to develop software with component requirements specification, provisioning, and assembling
 - d. all of the above
7. Explain why “Object orientation has failed but component software is succeeding.”
- a. Object orientation has a 20-year history.
 - b. Component-based approach is new and promising.
 - c. Object technology tends to ignore the aspects of economies and markets. Objects, by themselves, are not independently deployable. Components are independent units of deployment.
 - d. Object-oriented programming languages are too hard to use.
8. Explain why “Component standards are essential for component markets to develop and thrive.”
- a. With standards, we could build a general-purpose component.
 - b. A component standard specifies the necessary interfacing of certain components as is needed to allow clients and vendors to work together.
 - c. The interplay of components is not a problem of software engineering.
 - d. A component does not need many clients to be economically viable.
9. A common feature among SP, OOP, and COP is
- a. unification of data and function
 - b. divide and conquer for managing complexity
 - c. encapsulation for information hiding
 - d. interface specification
10. A unique feature for COP (which SP or OOP does not possess) is
- a. unification of data and function
 - b. divide and conquer for managing complexity
 - c. encapsulation for information hiding
 - d. interface specification
11. Which of the following has the highest composability, that is, the capability of one entity being integrated with other entity?
- a. SP
 - b. OOP
 - c. COP
 - d. XP (extreme programming)

Keys to Self-Review Questions

1. d 2. c 3. d 4. d 5. a 6. d 7. c 8. b 9. b 10. d 11. c

1.9 EXERCISES

1. When we discuss component-oriented programming, we require that a component is in binary form and to be independently deployable. Discuss why these requirements are essential for component-oriented programming.
2. Which of the following could be components and why?
 - a. A procedure
 - b. A class
 - c. A module
 - d. A function
 - e. An object
 - f. A subroutine
 - g. A UML diagram
 - h. A testing case
 - i. An event
 - j. Type declaration
 - k. C macros
 - l. C++ template
 - m. Smalltalk blocks
 - n. A protocol
 - o. A package in Java
 - p. An assembly in .NET
3. List three major goals for component-oriented programming.
4. List different forms of components along with brief explanations.
5. What are the two kinds of developing activities in component-based software engineering?

REFERENCES

- [Brown 1998] Brown, A. W. and Wallnau, K. C. “The current state of CBSE,” *IEEE Software*, Sept./Oct.: 37–46, 1998.
- [Cheesman 2001] Cheesman, J. and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.
- [Fischer 2002] Fischer, B. Deduction-Based Software Component Retrieval, Ph.D. thesis, <http://ase.arc.nasa.gov/people/fischer/papers/phd.html>, 2002.
- [Fukazawa 2002] Fukazawa, Y., Washizaki, H., and Yamamoto, H. Software Component Metrics, <http://www.fuka.info.waseda.ac.jp/Project/CBSE/metrics.html>, 2002.
- [Hoare 1969] Hoare, C. A. R. “An axiomatic basis for computer programming,” *Communications of ACM*, 12(10): 576–580, 1969.

- [Koza 1999] Kozaczynski, W. "Composite nature of component," *1999 International Workshop on Component-Based Software Engineering*, <http://www.sei.cmu.edu/cbs/icse99/papers>, 1999.
- [Mason 2002] Mason, D. V. Probabilistic Program Analysis for Software Component Reliability, Ph.D. thesis, www.sarg.ryerson.ca/~dmason/thesis.pdf, 2002.
- [Parrish 1999] Parrish, A., Dixon, B., and Hale, D. "Component based software engineering: a broad based model is needed," *1999 International Workshop on Component-Based Software Engineering*, <http://www.sei.cmu.edu/cbs/icse99/papers/16/16.htm>, 1999.
- [SIMS 2000] School of Information Management and Systems. University of California at Berkeley, How Much Information? <http://www.sims.berkeley.edu/research/projects/how-much-info/summary.html>, 2000.
- [Tanebaum 2001] Tanebaum, A. *The Modern Operating Systems*, 2nd ed., Prentice Hall, 2001.
- [Wang 2000] Wang, J. A. "Towards component-based software engineering," *Journal of Computing Sciences in Colleges*, 16(1): 177–189, 2000.
- [Yacoub 1999] Yacoub, S., Ammar, H., and Mili, A. "Characterizing a software component," *1999 International Workshop on Component-Based Software Engineering*, <http://www.sei.cmu.edu/cbs/icse99/papers/34/34.htm>, 1999.

2

THEORY OF COMPONENTS

Objectives of This Chapter

- Discuss principles of COP and its major features
- Investigate the infrastructures of COP technologies
- Provide a framework to unify various component technologies
- Provide formal definitions of software components
- Provide formal definitions of connections for component assembling
- Provide formal definitions of component deployment

2.1 PRINCIPLES OF COP

The word “component” has been around in computer industry for a long time. As a matter of fact, the concept of component itself had been living with us even before computers were invented. A house builder used components from several other industries to build a house. It is a common practice for automobile manufacturers to use many components from other industries to build a car. In computer hardware industry, engineers no longer design basic hardware elements from scratch for every product. Various microprocessor chips, memory chips, circuit boards, and network cards are available for building larger and powerful computing systems.

Even though we might have many different definitions about software components, the principles about software components remain the same for different definitions [Allen 1997; Garlan 2000; Liskov 2000; Luck 2000; Wang 2000, 2002]. Next, we will discuss some fundamental principles in component-based software engineering in general and in component-oriented programming in particular.

Principle 1: Components Represent Decomposition and Abstraction

The basic and effective strategy for tackling any large and complex problem in computer science is “divide and conquer.” One major idea in component-based software development is to create software modules that are themselves self-contained and independently deployable. Thus different developers will be able to work on different components independently, without needing much communication among themselves, and yet the components will work together seamlessly. In addition, during software maintenance phase, it will be possible to modify some of the components without affecting all of the others.

When we decompose a system, we factor it into separable components in such a way that

- each component is at the same level of detail;
- each component can be solved independently; and
- the implementations of these components can be integrated to satisfy the original system requirements.

Abstraction is a way to do decomposition productively by changing the level of detail to be considered. Software components hide certain details in an effort to provide only necessary information to the clients through their interface. The strategy of abstracting and then decomposing is typical in software development process. Decomposition is used to break software into components that can be combined to solve the original problem; abstraction assists in making a good choice of components. Computer science has gone through various abstractions. Procedural abstraction allows us to decompose a problem solution into independent functional units. Data abstraction or data type encapsulates data objects with a set of operations characterizing the behavior of the objects. This book is considering the third kind of abstraction: component abstraction, which is the highest level of abstraction in terms of its extension and useful information encapsulated.

Principle 2: Reusability Should Be Achieved at Various Levels

Software exists in different forms throughout the software engineering process. At the modeling and analysis phase, the requirements specification is seen as a form of software. In the design phase, architectural design and detail design documents are part of the software. Source code in the implementation phase and executable code deployed to the customer site are certainly software. Therefore, software reusability includes the reuse of any software artifacts in various formats.

As we discussed briefly in Chapter 1, there are five forms of software components, namely, component specification, component interface, component implementation, installed components, and component objects. Each form of the software components could be reused in different stages of a software life cycle.

Principle 3: Component-Based Software Development Increases the Software Dependability

With the rapid advances of computer hardware, highly reliable, powerful, and cheaper hardware are available for various applications. The dependability of a computing

system relies heavily on the trustworthiness of the software part. Component-based software development and component-oriented programming provide a systematical way to achieve dependable systems. Owing to the abstraction of components and systematic integration of components, it is much easier to validate critical requirements and verify safety for component-based systems. On the other hand, reusable components have usually been tested through the validation process and real usage for a long time and, therefore, their quality can be assured.

Principle 4: Component-Based Software Development Could Increase the Software Productivity

Component-based software is constructed by assembling existing reusable components rather than developing from scratch every time – reuse instead of reinventing the wheel. This process is much faster than developing an application from scratch in most cases.

Principle 5: Component-Based Software Development Promotes Software Standardization

As Clemens Czyperski described in [Clemens 2003], for component markets to develop, component standards must be in place. Standards can be used for creating an agreement on concrete interface specifications, enabling effective composition, and ensuring COP to be a new programming paradigm in which “plug-and-play” becomes a reality in software development just like the hardware counterpart.

2.2 INFRASTRUCTURES OF COP

We have discussed component definitions in Chapter 1, and we found out that it is not easy to have a unique component definition to fit in different situations. Such difficulties underlying the definition discussion are due to the following facts:

Software components are associated with their component infrastructure. Different component technologies have different component infrastructures, and thus have different component definitions.

What is a component infrastructure? Let us first find out the definition of *infrastructure*. According to [DIC 1995], infrastructure refers to the basic structures and facilities necessary for a country or an organization to function efficiently, for instance, buildings, transport, water, and energy resources, and administrative systems. The New Webster’s Encyclopedic Dictionary of the English Language [Random House Value Publishing, Inc., 1997] gives the following definitions for infrastructure: (1) the basic, underlying framework or features of a system or organization; (2) the fundamental facilities serving a country, city, or area, as transportation and communication systems, power plants, and roads; (3) the military installations of a country.

A component infrastructure is the basic, underlying framework and facilities for component construction and component management. It consists of three models: a component model, a connection model, and a deployment model. The component model defines what a valid component is and how to create a new component under

the component infrastructure. Component engineers build reusable components according to the component model. Each component infrastructure has a reusable component library containing building blocks confirming to the component model. The connection model defines a collection of connectors and supporting facilities for component assembling. Thus, the connection model determines how to build an application or a larger component out of existing components. The deployment model describes how to put components into a working environment.

The component infrastructure discussed here is sometimes called “component technology” or “component architecture” in literature. We prefer the name “component infrastructure” because it better serves our purposes in the component-oriented programming context. Component technologies complying with the above definition include JavaBeans from Sun Microsystems, COM (Component Object Model) and DCOM (Distributed COM) from Microsoft Corporation, and CORBA (Common Object Request Broker Architecture) from the Object Management Group. COM provides a framework for creating and using components on a Windows platform. It supports interoperability and reusability of distributed objects by allowing developers to build systems through assembling reusable components from different vendors that communicate via COM. It defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact, as long as the components adhere to the binary structure specified by Microsoft. DCOM extends COM to allow network-based components interaction. CORBA is a specification of a standard architecture, allowing vendors to develop ORB (Object Request Broker) products that support application portability and interoperability across different programming languages, hardware platforms, operating systems, and ORB implementations. A large and growing number of implementations of CORBA are available in the market place, including implementations from major computer manufactures and independent software vendors.

In the following six chapters, we will discuss six different component infrastructures, namely, JavaBeans, EJB, CORBA, .NET, OSGi, and Web Services. For each component infrastructure, we will discuss in detail its component model, connection model, and deployment model. We will see from the discussion that some component infrastructures are better than others in terms of their richness in one model and few component infrastructures are good in all these three models.

Component infrastructure has close relationship with algebra. Algebra is a mathematical object with internal structures. Let A be an algebra, which can be described as the following triple:

$$A = \langle D, C, L \rangle$$

where D is the domain of A , a set of objects under discussion within the algebra; C is a set of operators defined on the domain D ; and L is a set of laws describing the behaviors of objects with respect to the operators in C . For instance, in Boolean algebra, the domain $D = \{0, 1\}$, the operator set is $C = \{+, *, \sim\}$, and the law set L includes the following laws:

$$\begin{array}{lll} 0 + 0 = 0 & 0 * 0 = 0 & \sim 0 = 1 \\ 0 + 1 = 1 & 0 * 1 = 0 & \sim 1 = 0 \\ 1 + 0 = 1 & 1 * 0 = 0 & \\ 1 + 1 = 1 & 1 * 1 = 1 & \end{array}$$

Every component infrastructure embodies a component algebra. The domain of such an algebra is defined by the component model of the component infrastructure. The operator set of such an algebra is determined by the connection model of the component infrastructure. The laws of the algebra are abstractions of the deployment model of the component infrastructure.

Every component infrastructure is supported by a component builder tool or an IDE (integrated development environment). For instance, BDK (Bean Development Kit) was developed by Sun Microsystems as a visual design environment for Java beans, which supports JavaBeans component infrastructure. Microsoft Visual Studio .NET supports its .NET component infrastructure. JES (Java Embedded Server) 2.0 supports the OSGi component infrastructure. Most component builder tools support both “design for reuse” and “design with reuse” in component-based development. Some of them, however, have been designed only for “design with reuse.” For instance, BDK 1.1 does not have a source code editor, neither a compiler nor a JAR utility; thus, BDK 1.1 is not good for developing components, that is, developing for reuse. The user has to use another IDE to generate a Java bean packaged in a JAR file and load it into the “ToolBox” of BDK. Besides, BDK does not provide any method editor or event editor. Therefore, the use of BDK cannot modify a Java bean other than simply change some public properties. In terms of connection operators, BDK provides two operators to integrate components: modifying properties and event handling. With an ideal builder tool, a user should be able to generate a new component by either manually coding in a source code editor or by modifying an existing component taken from a reusable component library.

2.3 COMPONENT MODELS

Informally, a component has a collection of properties, methods, and events. Each component has a *Name*, an identifier presenting the component. *Properties* encapsulate states or attributes of a component. Each property has a type. *Methods* describe the behavior and services of a component. Each method has a signature:

```
visibility access return_type method_name (parameter-list).
```

Events describe the actions of which this component can initiate. Each event also has a type. *Interface*, a subset of the Cartesian set of the power sets of properties, methods, and events, defines the communication parts of the component to the outside world. We could use a graphical notation, called *component chart* (Figure 2.1), to present a component.

The component chart is a graphical representation of a software component. The rectangles, circles, and diamonds on both left and right sides represent public properties, methods, and events respectively in its interface. These interface elements act similarly as hardware pins around an integrated circuit chip. If there is no confusion from the context, we could just present one side of interface and omit the S (source) and T (target) part from the component chart, as shown in Figure 2.2 and Figure 2.3.

Formally, a component can be defined as

$$C = (P, M, E, I)$$

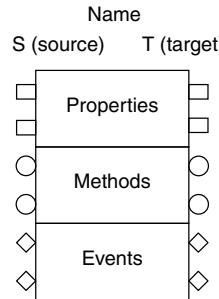


FIGURE 2.1. A component chart.

where

- $P = \{p:T \mid p \in \text{identifier}, T \in \text{type}\}$
- $M = \{(v,a,t,i(p)) \mid v \in \text{visibility}, a \in \text{access}, t \in \text{type}, i \in \text{identifier}, p \in \text{parameter}\}$
- $E = \{e:T \mid e \in \text{event}, T \in \text{type}\}$
- $I \subseteq 2^P \times 2^M \times 2^E$

The definitions for `identifier`, `type`, `visibility`, `access`, and `parameter` will be presented formally in Section 2.6.

The formal definition of component given here is only an abstraction of reusable software components. Different component infrastructures have their own concrete definitions for their components. The specification of a component also varies in different component infrastructures as well. For instance, the specification of a method in an interface can be given using pre- and postconditions, as illustrated in the following example.

Example 2.1 Suppose a component has a public method, called `squareRoot`, in its interface. The specification of this method could be given as below:

```
public static float squareRoot (float num)
```

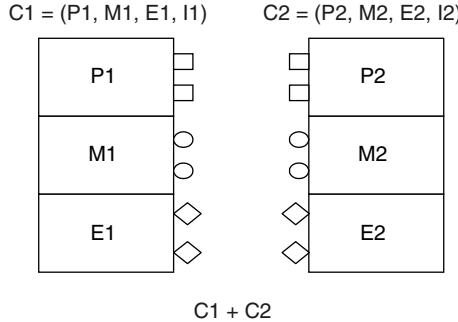
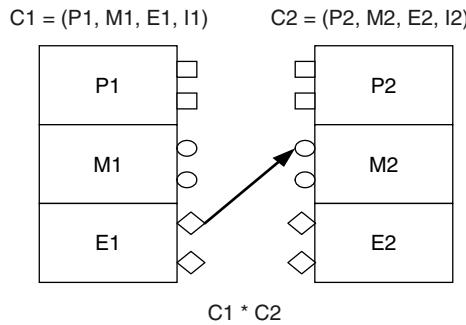
Pre-condition: `num > 0`

Post-condition: returns an approximation to the square root of `num`.

2.4 CONNECTION MODELS

Component software engineers can modify a component or integrate several components together to generate a new component or application with composition operators defined in a component infrastructure. In this section, we discuss some generic composition operators such as `+`, `-`, `*`, `/`, which are independent of any component infrastructures. Therefore, these operators can be interpreted differently in different component infrastructures.

1. *Aggregation or Addition (+)* Two components are added together without direct interactions, as shown in Figure 2.2.
2. *Reduction or Subtraction (-)* This is an inverse operator of addition.

**FIGURE 2.2.** Aggregation or addition of two components.**FIGURE 2.3.** Multiplication of two components.

3. *Event Handling or Multiplication (*)* Hook up an event from the source component to a method of the target component, as shown in Figure 2.3. When we need to express which event triggers the communication, we could attach the event to the operator as a subscript like: $C1 *_e C2$, where e is the triggering event for the hookup operation.
4. *Event to Property or Divide (/)* Hook up an event from the source component to a property of the target component. This operator is similar to multiplication except for the target effect. Similar to the case of multiplication, we could use this operator with a subscript like: $C1 /_e C2$, where e is the triggering event for the operation.
5. *Event to Event (^)* Hook up an event from the source component to an event of the target component. This operator is similar to multiplication except for the target effect. With this operator, an event e_1 from a component $C1$ could trigger an event e_2 in a component $C2$. This operation can be explicitly expressed as $C1 {}_{e_1}^{} e_2 C2$.
6. *Modification of Interface (+p, -p, +m, -m, +e, -e)* These operators are used to modify the interface of a component. For instance, the result of applying $+p$ to a component C is a component C' exactly same as C but with an explicit property p . The result of applying $-e$ to a component C is a component C' exactly same as C but without the event e in its interface.

Example 2.2 An Air-Conditioning System: This system contains three components: *AirCondition*, *Temperature*, and *Thermostat*. These three components are hooked up together with the operators discussed above. The *Thermostat* component has a property named *Comforting Temperature* that allows the user to modify at runtime. It has another property called *Current Temperature*, controlled by the *AirCondition* component instead of by user. *Temperature* component represents the current temperature provided by a temperature sensor. For demonstration purposes, users can change the value of current temperature at design time. Finally, the *AirCondition* component has three modes: AC OFF, COOLER, and HEATER. Its initial mode is AC OFF. When the current temperature is higher than the *Comforting Temperature*, COOLER is executed. In contrast, when the current temperature is lower than the *Comforting Temperature*, HEATER is executed. The architecture diagram of the system is illustrated in Figure 2.4.

Let C1, C2, and C3 in Figure 2.5 represent *AirCondition*, *Temperature*, and *Thermostat* respectively. These components have been implemented in Java and packaged as JavaBeans. P11, P12, P13, M11, M12, E11 are the properties, methods, and events in component C1 implemented as a JavaBean component, respectively, and similarly for C2 and C3.

There are three compositions in this system organized in a way presented in Figure 2.6. (1) C1*C2 is hooked up through an event from source component C1 to

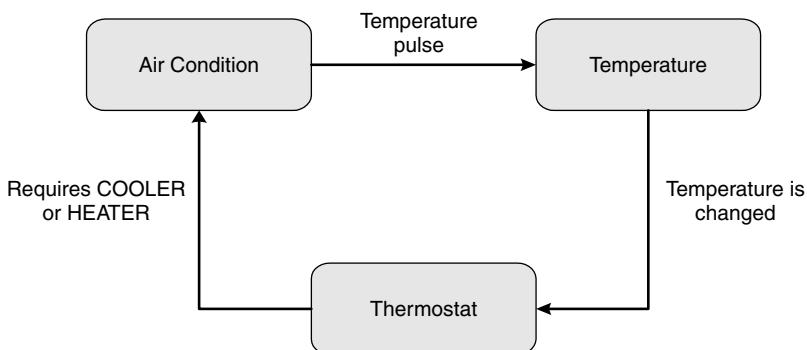


FIGURE 2.4. The air-conditioning system.

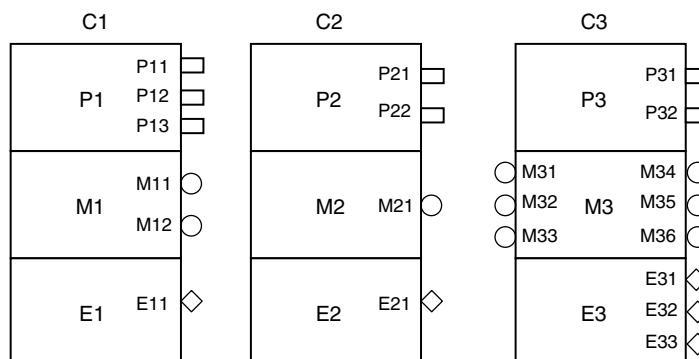


FIGURE 2.5. Components in the air-conditioning system.

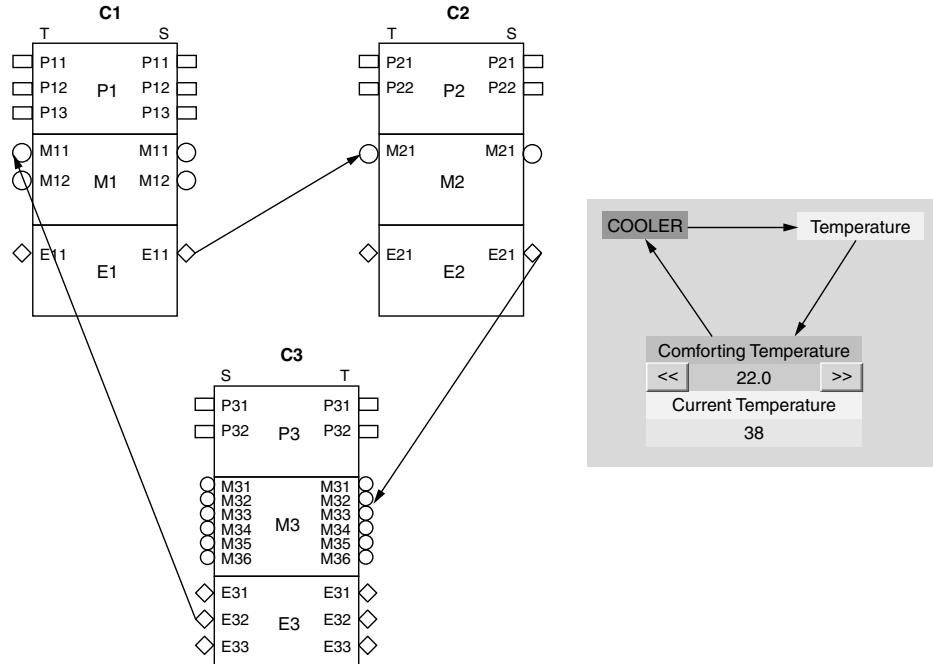


FIGURE 2.6. Component composition.

trigger a method in the target component C2. The event from C1 is E11 and the method in C2 is M21. (2) $C2*C3$ is hooked up through an event from source component C2 to trigger a method in target component C3. The source event from C2 is E21 and the method in C3 is M32. (3) $C3*C1$ is hooked up through an event from source component C3 to trigger a method in target component C1. The source event from C3 is E32 and the target method in C1 is M11. This three-way composition among the components in this design represents the structural relationship of the air-conditioning system, as illustrated in Figure 2.4 and Figure 2.6.

If we specify a component in text following the original quadruple definition, we obtain a *component table* that is similar to component charts in terms of their objectives. A component table expresses the component properties, methods, and events in a table format. In addition, a component table has a special slot to specify constraints of the component. These constraints are specified using first-order temporal logic formulas. This is useful for real-time and embedded systems, and it is important to include constraint specification capability for those components to capture nonfunctional user requirements as well as functional requirements. Since reactive systems interact with their environment without terminating, the partial correctness and termination are not adequate for their specification and verification [Pnueli 1992]. We need to be able to state and prove properties of execution sequences, not just pre- and postconditions. Therefore, we need temporal operators from first-order temporal logic to specify and verify temporal properties of embedded systems.

Example 2.3 A *TemperatureSet* component specified using a component table with temporal properties. The *TemperatureSet* component will be used as a control panel

for users to input the desired temperature setting or threshold temperature via a slider. This temperature will be sent to a *Switch* component to be used to compute the heater switch-state in a thermostat system. This component will use the class *ControlPanel* that will extend *JPanel* and implement *Serializable*. This component can also be used in other systems in which the user must set a threshold value via input hardware such as a knob or slider. Other systems may include heating/cooling, fuel tanks (determine the height in the tank), or cruise control for automobiles.

TemperatureSet
int currentValue
Public void setCurrentValue(int)
Public int getCurrentValue(void)
firePropertyChange
stateChanged
G(setCurrentValue(i) -> next(currentValue) = i)

This component has one property: `int currentValue` – the user-selected temperature from the slider object. It has two methods: `public void setCurrentValue (int)` – this method sets the `currentValue` property to the value of the slider – and the method `public int getCurrentValue(void)` that returns the value of the property `currentValue`. The event `firePropertyChange` will broadcast the updated value `currentValue` to all registered listeners. The event `stateChanged` will broadcast the updated value from the slider to all registered listeners. The special requirements slot of this component contains only one formula, which means that for all times in the future, if the method `setCurrentValue()` is called with an integer value `i`, then the thermostat control panel will display `i` as the current value. Here `G` is a temporal operator. `Gp` means that `p` is true at all times in the future. Other temporal operators include `F`, `U`, and `X`. The `F` operator is used to express a condition that must hold true at some time in the future. The formula `Fp` is true at a given time if `p` is true at some later time. The formula `pUq`, which is read “`p` until `q`,” means that `q` is eventually true, and until then, `p` must always be true. The formula `Xp` means that `p` is true at the next time. Notice that “`next`” is not a temporal operator. It is defined on state variables and thus it is a state operator.

The associations among components are represented graphically using arrows. There are two major associations: *message passing* and *event driving*. We use dash-line arrows to represent message passing and solid arrows to represent event driving relation. Figure 2.7 below illustrates message passing association.

The event driving associations are relationships among source components and target components. Source components are typical AWT or Swing components in Java programming environment, which can trigger an event. Target components provide event handlers for the event fired by the source components. Figure 2.8 depicts the event driving associations among three components.

**FIGURE 2.7.** Message passing associations.**FIGURE 2.8.** Event driving associations.

When *Button-1* is pressed, it triggers an *ActionEvent*. The event handler *actionPerformed()* is provided by the *Animation* component, similarly for the component *Button-2*.

2.5 DEPLOYMENT MODELS

The deployment model of a component infrastructure determines the process and activities of preparing a component for execution, including installation and any necessary configuration. For instance, EJB (enterprise Java Beans) produces a separate XML-based deployment descriptor to help deploy an EJB component. Web Services use UDDI (Universal Description, Discovery and Integration) as a key building block enabling enterprises to quickly and dynamically discover and invoke Web Services both internally and externally. Java-based component infrastructures deliver components as JAR files, while .NET components are called *assemblies*. The deployment model is one of the fundamental differences among different component infrastructures.

2.6 UNIFYING COMPONENT TECHNOLOGIES

In this section, we will define a formal specification language CSL (Component Specification Language) as a foundation for unifying different component technologies. We believe that a formal language for components is in need for several reasons:

- It serves to clarify confusions and misinterpretations.
- It is intellectually stimulating and challenging.
- A precise definition is a necessary condition for research and development in any scientific discipline.
- Formal models support formal specifications and formal verifications.
- Formal models support automatic tool development.

The syntax of the CSL is given in BNF-like notation. Terminal symbols are set in a typewriter font (*like this*). Non-terminals are set in an italic font (*like this*). The vertical bar | denotes an alternative in a rule. Parentheses (...) denote grouping. Parentheses with a trailing star sign (...)* denote zero, one, or several occurrences of

the enclosed item. Parentheses with a superscript plus sign $(\dots)^+$ denote one or several occurrences of the enclosed item. Brackets [...] denote an optional item.

Blanks

The following characters are considered as blanks: space, new-line, and horizontal tabulation. Blanks separate adjacent identifiers, literals, and keywords that would be otherwise confused as a single identifier, literal, or keyword. Apart from that, they are ignored.

Comments

Comments are Java-like comments. They start with two forward slashes “//” for a single-line comment. For a multiple line comment, it starts with /* and ends with */. Comments are treated as blanks.

Identifiers

Identifiers are a sequence of letters, digits, and _ (the underscore character) starting with a letter. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. Currently, we place no limit on the number of characters of an identifier.

$$\begin{aligned} \text{ident} &::= \text{letter} (\text{letter} \mid \text{digit} \mid _)^* \\ \text{letter} &::= \text{A} \dots \text{Z} \mid \text{a} \dots \text{z} \\ \text{digit} &::= \text{0} \dots \text{9} \end{aligned}$$

Integer Literals

An integer literal is a sequence of one or more digits. By default, integer literals are in decimal (radix 10).

$$\begin{aligned} \text{Integer-literal} &::= (0..9)^+ | \\ &\quad 0 \times (0..9|A..F|a..f)^+ | \\ &\quad 0o(0..7)^+ \end{aligned}$$

The following prefixes select a different radix:

Prefix	Radix
0x	Hexadecimal (radix 16)
0o	Octal (radix 8)

Boolean Literals

The *boolean* type has two possible values, represented by the literals `true` and `false`. A boolean literal is always of type boolean.

$$\begin{aligned} \text{boolean-literal} &::= \text{true} \mid \\ &\quad \text{false} \end{aligned}$$

Bit Literals

Bit literals are delimited by ‘ (single quote) characters.

```
bits-literal ::= ‘(bit)+
bit ::= 0|1|*|.
```

Prefix and Infix Operators

The following tokens are the CSL operators:

+	-	*	/	^	
+p	-p	+m	-m	+e	-e

Temporal and State Operators

The following temporal and state operators are used to capture special requirements for temporal properties:

```
Temporal_operator ::= G | F | U | X
State_operator ::= next() | previous()
```

G represents “always,” “henceforth,” or “global.”

F represents “eventually,” or “sometimes.”

U represents “until.” The formula pUq means that p keeps true before q becomes true.

X represents “next.” The formula Xp means that p is true at the next state.

The two state operators, next() and previous(), can only be applied on state variables.

Keywords

The identifiers below are reserved keywords:

```
extends     implements    requires    effects    public
protected   Private      static
```

Component Specification

```
Component ::= ident (Property, Method, Event, req) |  

Component + Component |  

Component - Component |  

Component * Component |  

Component/Component |  

Component^Component |  

+p Component | -p Component |  

+m Component | -m Component |  

+e Component | -e Component |  

Component || Component |
```

```

 $?(message) Component \ !(message) Component \ |$ 
Property ::=  $(ident\ type)^*$ 
Method ::=  $(visibility\ [access]\ type\ ident\ ([param]^*))^*$ 
Event ::=  $(ident\ type)^*$ 
visibility ::= public | protected | package | private
access ::= static | volatile
param ::= ident type
Type ::= int | String | any Java type
Req ::= Interface | p | Gp | Fp | pUq | Xp |
      next(ident) | previous (ident), where p and q are
      first-order logic formulas.

```

Formal Semantics of CSL

The semantics of CSL is based on many-sorted algebra. An algebra consists of a domain of values and a set of operations or functions defined on the domain: {set_of_values; operations}. Equations are given to define equivalences between syntactic elements; they specify the transformations that are used to translate from one syntactic form to another. The domain is often called a *sort*, and the domain and the function sections constitute the signature of the algebra. Functions with zero, one, and two operands are referred to as nullary, unary, and binary operations. If there is more than one domain, the algebra is called a *many-sorted algebra*. In Example 2.4, we have an abstract data-type *Stack* specified in a many-sorted algebra.

Example 2.4 Many-sorted algebra *Stack*

Domains:

Nat (the natural numbers), Stack (of natural numbers), and Bool (Boolean values).
 $N \in \text{Nat}$, $S \in \text{Stack}$

Functions:

newStack: () \rightarrow Stack
push: (Nat, Stack) \rightarrow Stack
pop: Stack \rightarrow Stack
top: Stack \rightarrow Nat
empty: Stack \rightarrow Bool

Axioms:

$\text{pop}(\text{push}(N, S)) = S$
 $\text{top}(\text{push}(N, S)) = N$
 $\text{empty}(\text{push}(N, S)) = \text{false}$
 $\text{empty}(\text{newStack}()) = \text{true}$

Errors:

$\text{pop}(\text{newStack}())$
 $\text{top}(\text{newStack}())$

In this example, a stack of natural numbers is modeled as a many-sorted algebra with three sorts (natural numbers, stacks, and Booleans) and five operations (newStack,

push, pop, top, empty). Next, we will define components with many-sorted algebra technique.

1. *Domains* Let S be the collection of all components, and let C_P , C_M , and C_E represent the set of properties, the set of methods, and the set of events of a component C respectively. Let IC_P , IC_M , and IC_E represent the set of properties, the set of methods, and the set of events of C in its interface respectively.

2. Functions

$$+ : S \times S \rightarrow S$$

$$C_1 + C_2 = C_3$$

$$C_{3k} = C_{1k} \cup C_{2k}, k \in \{P, M, E\}$$

$$IC_{3k} = IC_{1k} \cup IC_{2k}, k \in \{P, M, E\}$$

$$- : S \times S \rightarrow S$$

$$C_1 - C_2 = C_3, \text{ if } C_1 = C_2 + C_3$$

$$C_{3k} = C_{1k} \setminus C_{2k}, k \in \{P, M, E\}$$

$$IC_{3k} = IC_{1k} \setminus IC_{2k}, k \in \{P, M, E\}$$

$$C_1 - C_3 = C_2, \text{ if } C_1 = C_2 + C_3$$

$$C_{2k} = C_{1k} \setminus C_{3k}, k \in \{P, M, E\}$$

$$IC_{2k} = IC_{1k} \setminus IC_{3k}, k \in \{P, M, E\}$$

$$e^*m : S \times S \rightarrow S, , e \in C_{1E}, m \in C_{2M}$$

$$C_1 e^*m C_2 = C_3$$

$$C_{3P} = C_{1P} \cap C_{2P}$$

$$C_{3M} = C_{1M} \cap (C_{2M} - \{m\})$$

$$C_{3E} = (C_{1E} - \{e\}) \cap C_{2E}$$

$$IC_{3P} = IC_{1P} \cap IC_{2P}$$

$$IC_{3M} = IC_{1M} \cap (IC_{2M} - \{m\})$$

$$IC_{3E} = (IC_{1E} - \{e\}) \cap IC_{2E}$$

$$e/p : S \times S \rightarrow S, , e \in C_{1E}, p \in C_{2P}$$

$$C_1 e/p C_2 = C_3$$

$$C_{3P} = C_{1P} \cap (C_{2P} \setminus \{p\} \cup \{p'\}), \text{ where } p' \text{ is the updated property}$$

$$C_{3M} = C_{1M} \cap C_{2M}$$

$$C_{3E} = (C_{1E} - \{e\}) \cap C_{2E}$$

$$IC_{3P} = IC_{1P} \cap (IC_{2P} \setminus \{p\} \cup \{p'\}), \text{ where } p' \text{ is the updated property}$$

$$IC_{3M} = IC_{1M} \cap IC_{2M}$$

$$IC_{3E} = (IC_{1E} - \{e\}) \cap IC_{2E}$$

$$e1e2 : S \times S \rightarrow S, , e1 \in C_{1E}, e2 \in C_{2E}$$

$$C_1 e1e2 C_2 = C_3$$

$$C_{3P} = C_{1P} \cap C_{2P}$$

$$C_{3M} = C_{1M} \cap C_{2M}$$

$$C_{3E} = (C_{1E} - \{e1\}) \cap (C_{2E} - \{e2\})$$

$$IC_{3P} = IC_{1P} \cap IC_{2P}$$

$$\begin{aligned} IC_{3M} &= IC_{1M} \cap IC_{2M} \\ IC_{3E} &= (IC_{1E} - \{e1\}) \cap (IC_{2E} - \{e2\}) \end{aligned}$$

$$\begin{aligned} +p: S \rightarrow S \\ + p \quad C_1 = C_2 \\ C_{2P} &= C_{1P} \cup \{p\} \\ C_{2M} &= C_{1M} \text{ and } C_{2E} = C_{1E} \\ IC_{2P} &= IC_{1P} \cup \{p\} \\ IC_{2M} &= IC_{1M} \text{ and } IC_{2E} = IC_{1E} \end{aligned}$$

$$\begin{aligned} -p: S \rightarrow S \\ - p \quad C_1 = C_2 \\ C_{2P} &= C_{1P} \setminus \{p\} \\ C_{2M} &= C_{1M} \text{ and } C_{2E} = C_{1E} \\ IC_{2P} &= IC_{1P} \setminus \{p\} \\ IC_{2M} &= IC_{1M} \text{ and } IC_{2E} = IC_{1E} \end{aligned}$$

$$\begin{aligned} +m: S \rightarrow S \\ + m \quad C_1 = C_2 \\ C_{2M} &= C_{1M} \cup \{m\} \\ C_{2P} &= C_{1P} \text{ and } C_{2E} = C_{1E} \\ IC_{2M} &= IC_{1M} \cup \{m\} \\ IC_{2P} &= IC_{1P} \text{ and } IC_{2E} = IC_{1E} \end{aligned}$$

$$\begin{aligned} -m: S \rightarrow S \\ - m \quad C_1 = C_2 \\ C_{2M} &= C_{1M} \setminus \{m\} \\ C_{2P} &= C_{1P} \text{ and } C_{2E} = C_{1E} \\ IC_{2M} &= IC_{1M} \setminus \{m\} \\ IC_{2P} &= IC_{1P} \text{ and } IC_{2E} = IC_{1E} \end{aligned}$$

$$\begin{aligned} +e: S \rightarrow S \\ + e \quad C_1 = C_2 \\ C_{2E} &= C_{1E} \cup \{e\} \\ C_{2P} &= C_{1P} \text{ and } C_{2M} = C_{1M} \\ IC_{2E} &= IC_{1E} \cup \{e\} \\ IC_{2P} &= IC_{1P} \text{ and } IC_{2M} = IC_{1M} \end{aligned}$$

$$\begin{aligned} -e: S \rightarrow S \\ - eC_1 = C_2 \\ C_{2E} &= C_{1E} \setminus \{e\} \\ C_{2P} &= C_{1P} \text{ and } C_{2M} = C_{1M} \\ IC_{2E} &= IC_{1E} \setminus \{e\} \\ IC_{2P} &= IC_{1P} \text{ and } IC_{2M} = IC_{1M} \end{aligned}$$

$$\begin{aligned} ||: S \times S \rightarrow S \\ C_1 || C_2 = C_3 \end{aligned}$$

$$\begin{aligned}
C_{3P} &= C_{1P} \cap C_{2P} \\
C_{3M} &= C_{1M} \cap C_{2M} \\
C_{3E} &= C_{1E} \cap C_{2E} \\
IC_{3P} &= IC_{1P} \cap IC_{2P} \\
IC_{3M} &= IC_{1M} \cap IC_{2M} \\
IC_{3E} &= IC_{1E} \cap IC_{2E}
\end{aligned}$$

3. *Axioms* Let A, B, C be components; p a property; m a method; e an event. Let 0 represent the component whose property set, method set, and event set are all empty.

$$\begin{aligned}
A + B &= B + A \\
A^*B &\neq B^*A \\
A + A &= A \\
A + 0 &= A \\
(A + B) + C &= A + (B + C) \\
A^*(B + C) &= (A^*B) + (A^*C) \\
(A + B)^*C &= (A^*B) + (A^*C) \\
+p(-pA) &= A \quad -p(+pA) = A \\
+m(-mA) &= A \quad -m(+mA) = A \\
+e(-eA) &= A \quad -e(+eA) = A \\
+p(A + B) &= (+pA) + (+pB) \\
-p(A + B) &= (-pA) + (-pB) \\
+m(A + B) &= (+mA) + (+mB) \\
-m(A + B) &= (-mA) + (-mB) \\
+e(A + B) &= (+eA) + (+eB) \\
-e(A + B) &= (-eA) + (-eB) \\
+p(-pA) &= A = -p(+pA) \\
+m(-mA) &= A = -m(+mA) \\
+e(-eA) &= A = -e(+eA) \\
+p(A^*B) &= (+pA)^*(+pB) \\
-p(A^*B) &= (-pA)^*(-pB) \\
+m(A^*B) &= (+mA)^*(+mB) \\
-m(A^*B) &= (-mA)^*(-mB) \\
A || B &= B || A \\
(A || B) || C &= A || (B || C) \\
A || (B + C) &= (A || B) + (A || C) \\
(A + B) || C &= (A || B) + (A || C) \\
A + (B || C) &= (A + B) || (A + C) \\
(A || B) + C &= (A + C) || (A + C)
\end{aligned}$$

4. *Errors* All the operations other than defined above will generate errors or exceptions.

2.7 SUMMARY

Component-based development (CBD) has been recognized as an effective way to build software. However, there does not exist a widely accepted component definition

yet nor systematic component-based development methodology that can be applied throughout the software development process. A software component is a piece of self-contained code with well-defined functionality and can be reused as a unit in various contexts. From this definition, a component is a program or a collection of programs that can be compiled and made executable. A software component can be used and tested as a unit, independent of the context in which the component is eventually used. The internal implementation of a component is usually hidden from the user. The main premise behind a software component is the aspect of reuse. Software is best divided into components that are designed and implemented, each for a specific purpose. This allows the users or developers to partition the different aspects of a system into common areas that can be very distinctly identified, specified, designed, implemented, and ultimately be reused for other potential domain applications.

A component infrastructure is the basic, underlying framework and facilities for component construction and component management. It consists of three models: a component model, a connection model, and a deployment model. The component model defines what a valid component is and how to create a new component under the component infrastructure. Component engineers build reusable components according to the component model. Each component infrastructure has a reusable component library containing building blocks confirming to the component model. The connection model defines a collection of connectors and supporting facilities for component assembling. Thus, the connection model determines how to build an application or a larger component out of existing components. The deployment model describes how to put components into a working environment.

2.8 SELF-REVIEW QUESTIONS

1. The basic and effective strategy for tackling any large and complex problems in computer science is
 - a. artificial intelligence
 - b. distributed processing
 - c. parallel processing
 - d. divide and conquer
2. The word “component” has been used
 - a. in hardware engineering only
 - b. in software engineering only
 - c. in computer industry and other engineering disciplines for a long time
 - d. in the same meaning as the object or module
3. Abstraction is a way to do
 - a. programming and testing
 - b. decomposition productively by changing the level of detail to be considered
 - c. designing a system with object-oriented approach
 - d. integrating several components into a large application
4. Software reusability should be achieved at various levels because
 - a. software exists in different forms throughout the software life cycle

- b. software design has several steps
 - c. software implementation has different formats
 - d. software is ported into different platforms
- 5. Component-based software development increases the software dependability because
 - a. component-based software was developed using Java
 - b. component-based software has simpler architecture
 - c. reusable components have usually been tested through the validation process and real usage for a long time
 - d. reusable components have less overhead at run-time
- 6. Component-based software development could increase software productivity in
 - a. enterprise computing, where server-side programming plays a critical role
 - b. service-oriented computing, where nonfunctional requirements plays a critical role
 - c. “development for reuse,” in which a new component is developed and stored in a reusable library for future reuse
 - d. “development with reuse,” in which a new application is constructed by assembling existing reusable components
- 7. A component infrastructure refers to
 - a. the interface between components
 - b. the basic, underlying framework and facilities for components construction and management
 - c. the communication between components
 - d. the standard used to develop components
- 8. The component model defines
 - a. a collection of connectors and supporting facilities for component assembling
 - b. how to put components into a working environment
 - c. what a valid component is and how to create a new component under the component infrastructure
 - d. the interface to another component
- 9. The connection model defines
 - a. a collection of connectors and supporting facilities for component assembling
 - b. how to put components into a working environment
 - c. what a valid component is and how to create a new component under the component infrastructure
 - d. the interface to another component
- 10. There are different component definitions available because
 - a. component infrastructure has not been developed yet
 - b. software components are not associated with their component infrastructure
 - c. different component technologies have the same component infrastructure
 - d. different component technologies have different component infrastructures

Keys to Self-Review Questions

1. a 2. c 3. b 4. a 5. c 6. d 7. b 8. c 9. a 10. d

2.9 EXERCISES

1. Discuss some other principles of component-oriented programming that were not discussed in this chapter.
2. Describe component infrastructure with your own words. On the basis of your software development experience, list some software development environment similar to the component infrastructure defined in this chapter.
3. Draw a component chart for a *Button* component. It has the following properties: foreground color, background color, name, size, and image. It has a pair of methods (setter and getter) for each property above to set or get the property value. It has one event: `pressButton`.
4. Construct a component table for the *Button* component in the previous question.
5. Use component charts to describe the following component-based software system: A student information system consists of a user login component, a modifying component for faculty to upload student grades, and displaying component for students to check their grades on-line through a Web browser.
6. Write preconditions and post-conditions for a method `changeGrade(String student)` and a method `displayGrade()` in the student information system defined in the previous question.
7. How do you use UML diagrams to represent a software component?
8. How do you use UML diagrams to describe a component-based software system?
9. How do you describe the deployment model in a component infrastructure?
10. Discuss the strengths and weaknesses of UML as a modeling language to support the design of component-based software systems.

REFERENCES

- [Allen 1997] Allen, R. and Garlan, D. “A formal basis for architectural connection,” *ACM Transactions on Software Engineering and Methodology*, 6(3): 213–249, July, 1997.
- [DIC 1995] Hornby, A. S. *Oxford Advanced Learner’s Dictionary of Current English*, Oxford University Press, Oxford, UK, 1995.
- [Garlan 2000] Garlan, D., Monroe, R., and Wile, D. “ACME: Architectural description of component-based systems,” in *Foundations of Component-based Systems*, G. T. Leavens and M. Sitaraman (eds), Cambridge University Press, Cambridge, UK, 2000.
- [Liskov 2000] Liskov, B. and Guttag, J. *Program Development in Java, Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, Upper Saddle River, NJ, 2000.
- [Luck 2000] Luckham, D. C., Vera, J., and Meldal, S. “Key concepts in architecture definition languages,” in *Foundations of Component-based Systems*, G. T. Leavens and M. Sitaraman (eds), Cambridge University Press, Cambridge, UK, 2000.

- [Pnueli 1992] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, 1992.
- [Wang 2000] J. A. Wang, “Towards component-based software engineering,” *Journal of Computing Sciences in Colleges*, 16(1): 177–189, 2000.
- [Wang 2002] J. A. Wang, “Algebra for components,” in *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, Vol. 5, *Computer Science I*, N. Callaos, T. Leng, and B. Sanchez (eds), International Institute of Informatics and Systemics, 2002, pp. 213–218.

3

COP WITH JAVABEANS

Objectives of This Chapter

- Present an overview of JavaBeans technology
- Discuss the component infrastructure of JavaBeans
- Introduce the component model of JavaBeans
- Learn the connection model of JavaBeans
- Discuss the deployment model of JavaBeans
- Discuss the key features and techniques of component-oriented programming with JavaBeans

3.1 OVERVIEW OF JAVABEANS TECHNOLOGY

As we discussed in Chapters 1 and 2, a software component refers to a reusable, self-contained, independently deployable software unit. The JavaBeans specification [Sun 1997] defines software components, called *beans*, with an extra feature: beans are not only self-contained, reusable software units but are also visually constructed using builder tools like BDK [Sun1 2001] or Bean Builder [Sun2 2003]. Thus, JavaBeans components are good for designing graphical applications. With JavaBeans components, for instance, a new GUI (graphical user interface) can be built visually by “drag-and-drop” with minimal programming effort. The JavaBeans component infrastructure provides all the techniques for programmers to assemble larger applications from prebuilt, reusable JavaBeans components. JavaBeans components do not exist without such a component infrastructure, the underlying foundation or basic framework to construct, assemble, and deploy JavaBeans components.

There are a number of critical concepts for a JavaBeans component:

- *Properties* Public attributes of a bean that affect its appearance or behavior, for instance, background color, font, size, and so on.
- *Methods* Java methods that can be called from other beans or the environment.
- *Events* A source bean fires an event, while a listener bean receives the event and responds to the event.
- *Customization* Exposed properties could be modified at design time by a property editor or bean customizers.
- *Persistence* Enable a bean to save and restore its state.

The JavaBeans API includes interfaces and classes in the `java.beans` package and several other interfaces and classes from core Java API:

- The Java event model: `java.util.EventObject`, `java.awt.event`
- Object serialization: `java.io.Serializable`, `java.io.Object`
- Reflection: `java.lang.reflect`

In this chapter and in the next, we will study Java component architecture: JavaBeans and enterprise Java Beans (EJB), introducing component technology into Java programming language. While JavaBeans is mainly for client-side programming, EJB is designed for server-side programming. JavaBeans enables both software development *for reuse* and software development *with reuse*. It constitutes the fundamental platform for component-oriented programming in Java. In this chapter, we first introduce the basic concepts about JavaBeans, followed by a thorough discussion on the component model of JavaBeans. Various examples are given for creating a JavaBeans component from scratch or from an existing Java program. A JavaBeans component could be customized by modifying its properties. The connection model of JavaBeans allows us to assemble prebuilt beans into applets, stand-alone applications, or composite components. We will discuss the connection model using BDK 1.1 and Bean Builder 1.0. As for the deployment model in JavaBeans component infrastructure, bean components are packaged and delivered in JAR files that contain class files and supporting resources. We will discuss different deployment methods based on the two builder tools: BDK and Bean Builder.

3.2 COMPONENT MODEL OF JAVABEANS

3.2.1 Basic Concepts

According to [Sun 1997], “a JavaBean is a reusable software component that can be manipulated visually in a builder tool.” The original goal of JavaBeans was to “define a software component model for Java, so that third party ISVs (independent software vendors) can create and ship Java components that can be composed together into applications by end users” [Sun 1997]. JavaBeans API allows us to create reusable, platform-independent components.

Component-based software development intends to build large software systems by integrating prebuilt software components. The high productivity is achieved by using

standard components. The principles of component-based software development can be best described by the following two guiding principles: *reuse but do not reinvent (the wheel); assemble prebuilt components rather than coding line by line*. There are two basic activities in component-based software development. First, develop components for reuse. The production process model for this activity involves component specification, design, coding, testing, and maintenance. Second, develop software with existing components. A component search engine is used in this activity to obtain appropriate components while composing logic is applied to prove correctness of the component integration.

The JavaBeans architecture brings the component development model to Java. JavaBeans allows software developers to construct applications by piecing components together either programmatically or visually. Any Java program can be transformed into a Java bean. Since the entire runtime environment required by JavaBeans is part of the Java platform, no special libraries or classes have to be distributed with your components, which are fully portable to any platform supporting the Java runtime system. Since JavaBeans supports the features of software reuse and component models while it keeps all the benefits from Java, we could say that JavaBeans allows us to “Write once, run anywhere, and reuse everywhere.”

Although Java beans are intended to work in a visual application development tool, they do not necessarily have a visual representation at runtime. This means that beans must allow their property values to be changed through some type of visual interface and their methods and events should be exposed so that the development tool can write a code capable of manipulating the component when the application is executed. Creating a bean does not require any advanced concepts. Below is a piece of code that implements a simple bean:

```
public class MyBean implements java.io.Serializable
{
    protected int theValue;

    public MyBean () { }

    public void setMyValue( int newValue )
    {
        theValue = newValue;
    }
    public int getMyValue()
    {
        return theValue;
    }
}
```

This bean has the state (the instance variable `theValue`) that will automatically be saved by the JavaBeans persistence mechanism, and it has a property named `MyValue` that is usable by a visual programming environment. This bean does not have any visual representation, but that is not a requirement for a Java bean component. Several essential information and requirements for developing beans are listed below:

- All beans should implement the `Serializable` interface so that their state can be saved and later restored.

- *Methods* that are to be *exposed* to the builder tool and to other beans must be made *public*.
- All exposed methods should be made thread-safe (possibly *synchronized*) to prevent more than one thread from calling a method at any given time.
- *Properties* are exposed through the use of public “*set*” and “*get*” methods. Properties with no “*set*” method are *read-only*. Properties with no “*get*” method are *write-only*.
- The “*get*” side of a *boolean* property may be exposed either through the use of a public “*is*” method or an ordinary “*get*” method.
- Events that the bean can *multicast* are exposed through public “*add*” and “*remove*” methods.

Some important features and issues about JavaBeans are summarized below. Later we will revisit these concepts and discuss these issues in detail.

Properties, Methods, and Events

- *Properties* are attributes of a bean that are referenced by name. These properties are usually read and written by calling methods.
- The *methods* of a bean are just the Java methods exposed by the class that implements the bean. The methods represent the interface used to access and manipulate the component.
- *Events* are messages sent from one component to another, notifying the recipient that something interesting has happened.

The component sending the event is said to *fire* the event; the recipient is called a *listener*, and is said to *handle* the event. One component is free to listen to as many different events as it desires. On the other hand, many components can listen to a particular event; thus, components must be able to fire an event to an arbitrary number of components.

Introspection, Customization, and Persistence

- *Introspection* is the process of exposing the properties, methods, and events that a JavaBeans component supports.
- *Customization* is the process of modifying the attributes of a Java bean for specific purpose.
- *Persistence* refers to saving a bean component in its current state for future use. JavaBeans support the persistence model by implementing the `java.io.Serializable` interface. This interface helps save the customized bean by serializing the bean to a file.

Visibility, Multithreading, and Security

- *Visibility* is a runtime property of a bean. If a bean has a graphical user interface, it is *visible* at runtime in a builder tool. Even though there is no requirement that a bean be visible at runtime, it is necessary for a bean to support the visual application builder. An *invisible* runtime bean may provide custom property editors and customizers.

- *Multithreading* is the capability of a program to do several things at the same time. Java has multithreading capability, and so does JavaBeans. If a Java bean uses more than one thread during its execution, the programmer should follow all the principles of multithread programming. On the other hand, a Java bean should anticipate its use by more than one thread at a time and be able to handle the situation properly.
- *Security* is one of the most important features of Java. In general, we should assume that a bean is running in an untrusted applet. We should not make any design decisions that require our beans to be run in a trusted environment. All of the security restrictions apply to beans, such as denying access to the local file system, and limiting socket connections to the host system from which the applet was downloaded. Of course, if a bean is intended to run only in a Java application on a single computer, the Java security constraints do not apply.

One of the differences between a Java bean and an ordinary Java program is that a Java bean must be able to operate properly at both runtime and design time. A Java bean must be able to operate properly in a running application as well as inside an application development environment. At *design time*, the component must provide the design information necessary to edit its properties and customize its behavior. It also has to expose its methods and events so that the design tool can write code that interacts with the bean at *runtime*.

3.2.2 Creating a New Bean from a Java Program

Before reading this section, you should complete the Lab Practice 3.1: Install Java Beans Development Kit (BDK 1.1) on your computer.

Since Java beans are reusable components written in Java, in principle, every Java program can be transformed into a Java bean. Now the question is: What is the difference between a bean and an ordinary Java program? In other words, how to transform a Java program into a bean? Let us discuss the answers through an example.

Example 3.1 A Java program with GUI component JLabel in a JPanel:

```
//Exam3_1.java
//A simple JLabel in a JPanel container

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Exam3_1 extends JPanel {
    private JLabel l = new JLabel("A simple JLabel with raised
        border");
    private int width=200, height=100;

    public Exam3_1() {
        setSize(width, height);
        setBackground(Color.green);
```

```

        l.setFont(new Font("TimesNewRoman", Font.BOLD, 20));
        l.setForeground(Color.red);
        l.setBorder(BorderFactory.createRaisedBevelBorder());
        add(l);
    }

    public static void main(String args[]) {
        Exam3_1 ex = new Exam3_1();
        JFrame jf = new JFrame("Testing JLabel ... ");
        jf.getContentPane().add(ex, BorderLayout.CENTER);
        jf.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
        jf.setSize(ex.getPreferredSize().width + 20,
                  ex.getPreferredSize().height + 40);
        jf.setVisible(true);
    }
}

```

The output of the program is shown in Figure 3.1.

QUICK-REVIEW QUESTIONS

1. In the constructor of the class Exam3_1, the statement “`setSize(width, height);`” is not necessary for this example, why?
2. At the end of `Exam3_1.java`, we set the size of the `JFrame` to be the “preferred size” of the `JPanel` plus some offset pixels. What is the *preferred size* of a GUI component?
3. What is the difference between the two `Component` methods: `show(true)` and `setVisible(true)`?

The `Component` methods `show()` and `show(boolean)` were defined in JDK1.1. They were deprecated and were replaced by `setVisible(boolean)` since JDK1.2.

Every GUI component in Java has a *preferred size*. The preferred size is generally the smallest size necessary to render the component in a visually meaningful way.

Now let us see how to transform the Java program `Exam3_1.java` into a Java bean.

Step 1: Use package Statement as the First Line of Your Source Code Normally, classes that represent a bean are first placed into a package. It is especially useful



FIGURE 3.1. The output of `Exam3_1.java`: `JLabel`.

to generate a JAR file for a bean with a number of multimedia files. Even for this extremely simple example, we use a package statement as its first line as below:

```
package Pack3_2;
```

This means that we are going to create a package (actually a directory) called Pack3_2, and all the information necessary for deploying the bean will be put there.

Step 2: Implement the Serializable Interface for Persistence At the line of class definition, make your class implement the **Serializable** interface. This is to support persistence – saving a bean object in its current state for future use. All the Java beans should implement the **Serializable** interface to support persistence. Note that the **Serializable** interface belongs to the `java.io` package, and usually we import this package at the beginning of our programs.

Below is a list of the complete source code for the Java bean adapted from Exam3_1.java. The only two changes are **highlighted**.

Example 3.2 A bean converted from a Java program with a JPanel superclass:

```
//Exam3_2.java
//A simple JLabel bean

package Pack3_2;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class Exam3_2 extends JPanel implements Serializable {

    private JLabel l = new JLabel("A simple JLabel with raised
        border");
    private int width=200, height=100;

    public Exam3_2() {
        setSize(width, height);
        setBackground(Color.green);
        l.setFont(new Font("TimesNewRoman", Font.BOLD, 20));
        l.setForeground(Color.red);
        l.setBorder(BorderFactory.createRaisedBevelBorder());
        add(l);
    }

    public static void main(String args[]) {
        Exam3_2 ex = new Exam3_2();
        JFrame jf = new JFrame("Testing JLabel ... ");
        jf.getContentPane().add(ex, BorderLayout.CENTER);
        jf.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
```

```
        System.exit(0);
    }
});
jf.setSize(ex.getPreferredSize().width + 20,
           ex.getPreferredSize().height + 40);
jf.setVisible(true);
}
```

Step 3: Compile Your Packaged Classes Using the `-d` Option The line command used for compiling this example is the following:

```
javac -d . Exam3 2.java
```

Note that the `-d <directory>` option for the Java compiler means that we are going to put the compiled classes in the directory `<directory>`. In this example, we put all generated class files in the current directory (the `“.”`). Since we used a statement of `“package Pack3_2;”` in our source code, all the class files are actually put into the `.\Pack3_2` directory.

Step 4: Create a Text File to Describe the Contents of a JAR File The BDK requires the beans to be in a JAR (Java Archive) file. JAR files are used for packaging related class files, serialized beans, and other resources. We will use JAR files to bundle beans and their supporting classes. As a matter of fact, JAR files have many other applications including the improvement of download performance of Java applets in a Web page.

The manifest file tells the `jar` program to generate JAR files as desired. It contains a series of attribute/value pairs specifying various attributes of the contents of the archive. We will mainly use the manifest file to specify whether a particular class is a Java bean and which class is the main class in this archive. In our example, we name our manifest file as `manifest3_2.tmp`. The contents of `manifest3_2.tmp` is shown below:

Main-Class: Pack3_2.Exam3_2
Name: Pack3_2/Exam3_2.class
Java-Bean: True

Note that the syntax of the `manifest` file must be correct:

- Each class listed in the manifest file should be separated from all other classes by a blank line.
 - If the class is a bean, its Name: line should be immediately followed by the line of Java-Bean: True.
 - If the class is not a bean, its Name: line should be immediately followed by the line of Java-Bean: False.
 - The Main-Class: line uses dots (.) to separate package names and class names.
 - The Name: line uses forward slash (/) to separate package names and class names.

Step 5: Create The JAR File for Your Bean Using The jar Utility For this example, we use the following command to generate our JAR file:

```
jar cfm Exam3_2.jar manifest3_2.tmp Pack3_2\*.*
```

- The option `cfm` means that
 - We are going to create (`c`) a JAR file (`f`), and
 - this JAR file is named as the first argument (`Exam3_2.jar`), and
 - the manifest (`m`) file is specified in the second argument (`manifest3_2.tmp`).
- the third argument (`Pack3_2*.*`) indicates that all the files in the `\Pack3_2` directory should be included in the JAR file.
- the directory structure in the JAR file must match the directory structure used in the `manifest` file. Thus, we need to execute the `jar` command from the directory in which `Pack3_2` is located.

Step 6: Check If The Files Were Archived Correctly For this example, we can use the following command to check whether the JAR file was generated as expected:

```
jar tvf Exam3_2.jar | more
```

This command will list all files in the JAR file `Exam3_2.jar` in table form as shown below:

```
D:\Book\COP\Ch3>jar tvf Exam3_2.jar
 0 Thu Jun 8 11:00:22 CDT 2000 META-INF/
 145 Thu Jun 8 11:00:22 CDT 2000 META-INF/MANIFEST.MF
 387 Thu Jun 8 10:59:52 CDT 2000 Pack5_2/Exam5_2$1.class
1744 Thu Jun 8 10:59:52 CDT 2000 Pack5_2/Exam5_2.class
```

Here the `META-INF` directory and the `MANIFEST.MF` file were generated by the `jar` program automatically.

Step 7: Test Your Java Bean Wrapped in a JAR File This can be done by using `java` utility with the `-jar` option. For this example, we use the following command:

```
java -jar Exam3_2.jar
```

Step 8: Add the Bean into The BeanBox There are two ways to do this:

1. Place the JAR file in the `BDK1.1\jars` directory.
2. Use the BeanBox `file` menu's `LoadJar ...` option to locate the JAR file on your system and load it into the BeanBox's ToolBox.

We summarize the discussion above into the following Code Pattern and 8-step table:

```
package Pack_name;

import java.awt.*;
import java.awt.event.*;
```

```

import javax.swing.*;
import java.io.*;

public class MyBean extends JPanel implements Serializable {
    //creating MyBean
    java_statements;
}

```

Code 3.1. Creating a Java Bean

The eight steps to create a simple Java bean is summarized below using Exam3_2.java as an example:

Step 1:	Use package statement as the first line of your source code. package Pack3_2;
Step 2:	Implement the Serializable interface for persistence. public class Exam3_2 extends JPanel implements Serializable
Step 3:	Compile your packaged classes using the -d option. javac -d . Exam3_2.java
Step 4:	Create a manifest file to describe the contents of a JAR file. edit manifest3_2.tmp
Step 5:	Create the JAR file for your bean using the jar utility. jar cfm Exam3_2.jar manifest3_2.tmp Pack3_2*.*
Step 6:	Check if the files were archived correctly. jar tvf Exam3_2.jar more
Step 7:	Test your Java bean wrapped in a JAR file. java -jar Exam3_2.jar
Step 8:	Add the bean into the BeanBox.

Note that these steps are applicable to create a new bean from an existing Java program. If you develop a bean from scratch, then Step 7 above should be omitted.

QUICK-REVIEW QUESTIONS

- If we delete the main method in Exam3_2.java and repeat the eight steps discussed above, what will happen? Can we obtain a new bean anyway?
- If we have a Java application whose direct superclass is JFrame (instead of JPanel as in Example 3.2), can we add the bean converted from the Java program into the BeanBox?

3. If we have a Java applet, how do we convert it into a bean? Can we add the result bean into the BeanBox?

Example 3.3 A Java application with a JFrame superclass:

```
//Exam3_3.java
//A Java application with JFrame as its superclass

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Exam3_3 extends JFrame {
    private JLabel l = new JLabel("Hello, this is a testing JFrame
        ...");
    private Container c = getContentPane();

    public Exam3_3() {
        super("Exam3_3 - JFrame");
        c.setLayout(BorderLayout.CENTER);
        setSize(l.getPreferredSize().width+50,
l.getPreferredSize().height+50);
        setVisible(true);
    }

    public static void main(String args[]) {
        Exam3_3 f = new Exam3_3();
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

The output of this program is shown in Figure 3.2.
The Java program Exam3_3.java can be converted into a Java bean in the following example:



FIGURE 3.2. The output of Exam3_3.java.

Example 3.4 A bean converted from a Java program with a JFrame superclass:

```
//Exam3_4.java
//A bean Converted from a Exam3_3.java with a JFrame superclass
package Pack3_4;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Exam3_4 extends JFrame implements Serializable {
    private JLabel l = new JLabel("Hello, this is a testing
        JFrame ...");
    private Container c = getContentPane();

    public Exam3_4() {
        super("Exam3_4: Converted from Exam3_3");
        c.add(l, BorderLayout.CENTER);
        setSize(l.getPreferredSize().width+50,
l.getPreferredSize().height+50);
        setVisible(true);
    }
}
```

The manifest file used for this example is listed below:

Main-Class: Pack3_4.Exam3_4

Name: Pack3_4/Exam3_4.class

Java-Bean: True

After creating the JAR file, we can successfully load the JAR file into the ToolBox of BDK. However, we could not insert the bean (`Exam3_4.jar`) into the BeanBox. This problem is due to the containment hierarchy of GUI components. Since the BeanBox is a Frame, it cannot hold another Frame. The conclusion of the discussion can be restated as the following:

If you want to create a Java bean that can be manipulated inside the BeanBox, its superclass should be a Component other than Frame or JFrame.

3.2.3 Creating a New Bean from Scratch

In this section, we will investigate how to create Java beans from scratch.

3.2.3.1 Invisible Java Beans If a Java bean does not have a graphical user interface, it will not be “visible” inside the BeanBox. An invisible bean actually shows a name in plain text when you drop it into the BeanBox. You can hide invisible beans completely from showing in the BeanBox by selecting “Hide Invisible Beans” from the menu “View” of the BeanBox window. There are no properties shown in the Property window for invisible beans.

Example 3.5 A simple invisible bean:

```
//Exam3_5.java
//An invisible bean with no GUI

import java.io.*;

public class Exam3_5 implements Serializable {
    public static void main (String args[]) {
        System.out.println("This is an invisible Java bean.");
    }
}
```

We first compile this program using the following line command:

```
javac Exam3_5.java
```

Then, we edit a text file named manifest3_5.tmp with the following contents:

```
Main-Class: Exam3_5

Name: Exam3_5.class
Java-Bean: True
```

We create a JAR file using the following command:

```
jar cfm Exam3_5.jar manifest3_5.tmp Exam3_5.class
```

Then, we can load the JAR file into the BeanBox. The bean shows its main class name as "Exam3_5" in plain text in the BeanBox. We call it "invisible" because it does not have a graphical interface and it does not show any properties in the Property window. It does not show any events either. We can hide it completely from the BeanBox by selecting the BeanBox menu "View" and then "Hide Invisible Beans".

QUICK-REVIEW QUESTIONS

1. The bean Exam3_5 is invisible because it does not have a graphical user interface.
Are all beans with graphical user interfaces *visible* beans?
2. The bean Exam3_5 has zero visible property in the Property window, and no visible event can be fired from this bean. However, it has three visible methods displayed in the EventTargetDialog window:

`equals, notify, notifyAll.`

Where are they coming from?

3. If we change the java interpreter command in beanbox/run.bat to

```
java sun.beanbox.BeanBoxFrame > beanreport.txt
```

we can receive a report file on information associated with a bean. Now, load Exam3_5.jar into the BeanBox and select menu “Edit” - “Report . . . ” and check the file bean-report.txt after exiting from BeanBox, we will see all 10 public methods are listed! Why were only 3 methods displayed in the EventTargetDialog window?

A bean might be invisible even it uses graphical user interface. Below, we present one example.

Example 3.6 A simple invisible bean with GUI:

```
//Exam3_6.java
//An invisible bean with GUI

import javax.swing.JOptionPane;
import java.io.Serializable;

public class Exam3_6 implements Serializable {
    public static void main(String args[]) {
        JOptionPane.showMessageDialog( null, "This is a
            \nJOptionPane Java bean.");
        System.exit(0);
    }
}
```

This bean is invisible even when it has graphical user interface. This bean shows no visible property nor visible event. Like Exam3_5, however, it has 3 visible methods displayed in the EventTargetDialog window:

equals, notify, and notifyAll.

This example indicates that an invisible bean does not necessarily have no graphical user interface. An invisible bean is a Java bean not inheriting from a Component or JComponent, thus showing no visible property in the Property window.

3.2.3.2 Beans Inherited from Canvas The Canvas class inherits from java.awt.Component. A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user. A bean inheriting from Canvas can be very simple, as shown in the Example 3.7 below. The bean should override the paint method in order to draw custom graphics on the canvas.

Example 3.7 A bean inheriting from a Canvas:

```
//Exam3_7.java
//A simple bean with superclass Canvas

import java.awt.*;
import java.io.*;

public class Exam3_7 extends Canvas implements Serializable {
    public Exam3_7() {
```

```

        setSize(60, 50);
    }

    public void paint(Graphics g) {
        g.drawOval(5,5,50,30);
        g.drawOval(12,12,15,8);
        g.drawOval(30,12,15,8);
        g.fillOval(19,12,8,8);
        g.fillOval(37,12,8,8);
        g.drawArc(22,20,16,8,180,180);
        g.drawString("Hello!", 16, 45);
    }
}

```

After compiling it, a JAR file can be generated from a simple manifest file as shown below:

```

Main-Class: CanvasBean1
Name: CanvasBean1.class
Java-Bean: True

```

Figure 3.3 shows a window capture of the bean Exam3_7 after loading it into the BeanBox.

There are four visible properties shown in the Property window: **background**, **foreground**, **name**, and **font**. There are 22 visible events corresponding to 9 kinds of event handlers, as shown in Figure 3.4.

There are 19 visible methods displayed in the EventTargetDialog window: **dispatchEvent**, **equals**, **addNotify**, **disable**, **doLayout**, **enable**, **hide**, **invalidate**, **layout**, **list**, **nextFocus**, **notify**, **notifyAll**, **removeNotify**, **repaint**, **requestFocus**, **show**, **transferFocus**, **validate**.

QUICK-REVIEW QUESTIONS

1. In the program Exam3_7.java, we did not define those properties, events, and methods listed above. Where do they come from?
2. Some of the properties are not very useful for certain purposes, for instance, the “name” property of the bean Exam3_7 for the visual manipulation of the bean.



FIGURE 3.3. The window capture for the bean Exam3_7.

Number	Event	Event Handler
1	Property	propertyChanged(e)
2	Component	componentResized(e)
3	Component	componentMoved(e)
4	Component	componentShown(e)
5	Component	componentHidden(e)
6	Mouse	mouseMoved(e)
7	Mouse	mouseDragged(e)
8	Mouse	mousePressed(e)
9	Mouse	mouseReleased(e)
10	Mouse	mouseClicked(e)
11	Mouse	mouseExited(e)
12	Mouse	mouseEntered(e)
13	Hierarchy	hierarchyChanged(e)
14	Key	keyTyped(e)
15	Key	keyPressed(e)
16	Key	keyReleased(e)
17	Focus	focusGained(e)
18	Focus	focusLost(e)
19	Hierarchy	ancestorMoved(e)
20	Hierarchy	ancestorResized(e)
21	InputMethod	inputMethodTextChanged(e)
22	InputMethod	caretPositionChanged(e)

FIGURE 3.4. Visible events for the bean inheriting from Canvas.

How do we stop them showing in the Property window? Similarly for events and methods, how do we control them so that only those “useful” for our purpose are revealed to outside?

3. Can we add GUI components such as JButton or JList into a Canvas?

All those properties, events, and methods listed above are actually inherited from the superclass `Canvas`, which again inherits from `Component`.

We summarize this section by the following Code Pattern for a bean with superclass `Canvas`:

```

package Pack_name;

import java.awt.*;
import java.io.*;

public class BeanName extends Canvas implements Serializable {
    public BeanName () {
        setSize(60, 50);
    }

    public void paint(Graphics g) {
        g.drawString("Hello!", 16, 45);
        //other drawing statements
    }
}

```

Code 3.2. Creating a Java Bean Inheriting from Canvas

3.2.3.3 Beans Inherited from Panel or JPanel Panel is the simplest container class. A panel provides space in which an application can attach any other component, including other panels. The default layout manager for a panel is the FlowLayout layout manager.

Example 3.8 A bean inheriting from JPanel:

```

//Exam3_8.java
//A simple bean with superclass JPanel

import javax.swing.*;
import java.io.*;

public class Exam3_8 extends JPanel implements Serializable {
    private JLabel l = new JLabel("A Bean inheriting from
        JPanel");

    public Exam3_8() {
        setSize(getPreferredSize());
        add(l);
    }
}

```

Figure 3.5 shows a window capture of the bean Exam3_8 after loading it into the BeanBox.

There are 11 visible properties shown in the Property window: doubleBuffered, opaque, autoscrolls, background, alignY, alignX, debugGraphicsOptions, foreground, requestFocusEnabled, font, and verifyInputWhenFocusTarget.

There are 28 visible events corresponding to 12 kinds of event handlers, as shown in Figure 3.6. In addition to the 22 visible events corresponding to 9 kinds of event



FIGURE 3.5. The window capture for the bean Exam3_8.

handlers as shown in Figure 3.4, the beans inheriting from `JPanel` add the following 6 events corresponding to 3 event handlers.

There are 25 visible methods displayed in the EventTargetDialog window: In addition to the 19 methods listed for `Canvas` (`dispatchEvent`, `equals`, `addNotify`, `disable`, `doLayout`, `enable`, `hide`, `invalidate`, `layout`, `list`, `nextFocus`, `notify`, `notifyAll`, `removeNotify`, `repaint`, `requestFocus`, `show`, `transferFocus`, `validate`), `JPanel` adds the following 6 methods: `getClientProperty`, `grabFocus`, `removeAll`, `resetKeyboardActions`, `revalidate`, and `updateUI`.

3.2.3.4 Beans Inherited from Applet or JApplet

Example 3.9 A bean inheriting from a `JApplet`:

```
//Exam3_9.java
//A simple bean with superclass JApplet

import java.awt.*;
import javax.swing.*;
import java.io.*;

public class Exam3_9 extends JApplet implements Serializable {
    private JLabel l = new JLabel("A Bean inheriting from
        Japplet");
    private JButton b = new JButton("Test JButton");

    public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(l);
        c.add(b);
    }
}
```

Figure 3.7 shows a window capture of the bean Exam 3-9 after loading it into the BeanBox. The visible properties, events, and methods of this bean are similar to the bean inheriting from `Canvas` (Example 3.7). We leave its details as an exercise.

3.2.3.5 Adding an Icon to a Bean Some of the beans in the ToolBox have icons attached to them. If you want to add an icon to your bean, you need to prepare a small

Number	Event	Event Handler
1	Property	propertyChanged(e)
2	Component	componentResized
3	Component	componentMoved(e)
4	Component	componentShown(e)
5	Component	componentHidden(e)
6	Mouse	mouseMoved(e)
7	Mouse	mouseDragged(e)
8	Mouse	mousePressed(e)
9	Mouse	mouseReleased(e)
10	Mouse	mouseClicked(e)
11	Mouse	mouseExited(e)
12	Mouse	mouseEntered(e)
13	Hierarchy	hierarchyChanged(e)
14	Key	keyTyped(e)
15	Key	keyPressed(e)
16	Key	keyReleased(e)
17	Focus	focusGained(e)
18	Focus	focusLost(e)
19	Hierarchy	ancestorMoved(e)
20	Hierarchy	ancestorResized(e)
21	InputMethod	inputMethodTextChanged(e)
22	InputMethod	caretPositionChanged(e)
23	Container	componentAdded(e)
24	Container	componentRemoved(e)
25	VetoableChange	vetoableChanged(e)
26	Ancestor	ancestorMoved(e)
27	Ancestor	ancestorAdded(e)
28	Ancestor	ancestorRemoved(e)

FIGURE 3.6. Visible events for the bean inheriting from JPanel.



FIGURE 3.7. The window capture of the bean Exam3_9.

graphical icon (16 pixels in width and height) first and then develop a Java program to show this icon when the JAR file is loaded into the ToolBox.

Example 3.10 A BeanInfo class for attaching an icon to a bean:

```
//Exam3_10BeanInfo.java
//The BeanInfo class for the bean inheriting from Canvas
//Just display an icon

package Pack3_10;

import java.beans.*;
import java.awt.*;

public class Exam3_10BeanInfo extends SimpleBeanInfo {
    public Image getIcon(int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_16x16) {
            Image img = loadImage("wja.gif");
            return img;
        }
        if (iconKind == BeanInfo.ICON_COLOR_32x32) {
            Image img = loadImage("wja.gif");
            return img;
        }
        return null;
    }
}
```

The Java program is called *BeanInfo class* because it is a supporting class that provides information about the bean. Note the public class name must be the bean name followed by “BeanInfo” suffix. It inherits from `SimpleBeanInfo` class that implements `BeanInfo` interface in the `java.beans` package. The `SimpleBeanInfo` method `loadImage()` takes an image file name as its parameter and returns an `Image` object. By default, the `SimpleBeanInfo` method `getIcon()` claims that there are no icons available for the bean. Here we overrode the `getIcon` method so that it returns an `Image` object as the requested icon. This program should be compiled and its class file should be included into the JAR file along with the image file `wja.gif`. After loading this JAR file into the ToolBox window, you will see a small icon that appears to the left side of the bean name `Exam3_10`, as shown in Figure 3.8.

We summarize the discussion about `BeanInfo` class by presenting the following Code Pattern:



FIGURE 3.8. The bean Exam3_10 with an icon.

```

package Pack_name;

import java.beans.*;
import java.awt.*;

public class BeanNameBeanInfo extends SimpleBeanInfo {
    public Image getIcon(int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_16x16) {
            Image img = loadImage("wja.gif");
            return img;
        }
        if (iconKind == BeanInfo.ICON_COLOR_32x32) {
            Image img = loadImage("wja.gif");
            return img;
        }
        return null;
    }
}

```

Code 3.3. A BeanInfo Class to Load an Icon

QUICK-REVIEW QUESTIONS

1. For the BeanInfo class discussed in Example 3.10, what file name must it have? What name must a BeanInfo class source file have generally?
2. What other functions does the BeanInfo class provide in addition to load an icon to a bean?
3. In the current version of BDK, you cannot modify or update a bean after you load it into the BeanBox, unless you exit from the BeanBox, modifying the JAR file, restarting the BeanBox, and loading the JAR file again. Is there any way to go around this problem?

3.2.4 Customizing a Java Bean

A Java bean can be customized in many different ways. For instance, we can add or remove some properties for a bean, similarly for events and methods revealed to the bean users.

3.2.4.1 Adding Properties to a Bean For a simple bean like Exam3_10 discussed in the last section, it displays only four properties in the property sheet demonstrated in the Figure 3.9: font, name, background, and foreground. The user can customize the bean by changing the values of these properties. For instance, the foreground and background color can be changed to whatever you like. The font style and font size for the text string “Hello” can also be changed. However, you cannot change the text string itself because the property sheet does not provide such a property.

We can add a new property, say “text,” into the property sheet of this bean by adding a property *set* method and a property *get* method code to the source file, following some Code Patterns shown below:

```
public void setText(String s)
public String getText()
```

In general, to add a property called *PropertyName*, we must follow the following Code Pattern:

```
//For non-boolean type property:

    public void setPropertyName (DataType value)

    public DataType getPropertyName ()

//For boolean type property:

    public void setPropertyName (boolean value)

    public boolean isPropertyName ()
```

Code 3.4. Adding a Property to a Bean (Source Code)

Here we emphasize some points for the Code Pattern:

- The *set* method returns *void* and takes one argument. On the other hand, the *get* method returns the same type as the corresponding *set* method’s argument.
- The first letter of property name is capitalized in the *set/get* method names.

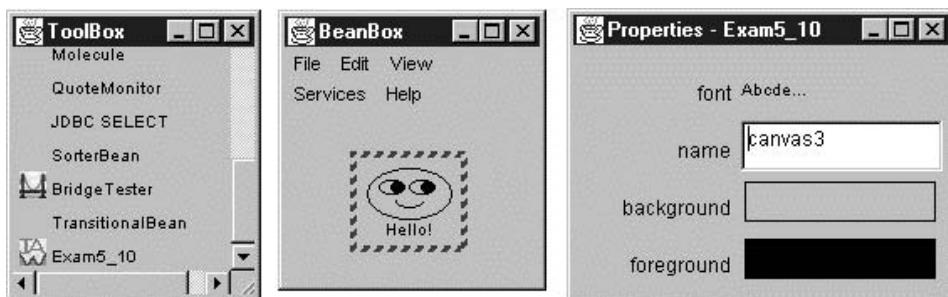


FIGURE 3.9. The property sheet for the bean Exam3_10.

- For boolean type property, the *get* method name begins with the word *is* rather than *get*.

The Bean Builder tool uses an introspection process to examine a bean. When it finds a set/get method pair that matches the Code Pattern 3.4, it exposes the corresponding property in the property sheet of the bean.

Example 3.11 Adding more properties to the bean Exam3_10:

```

1  //Exam3_11.java
2  //Based on Exam3_10 with superclass Canvas, more properties
3  added
4
5  package Pack3_11;
6
7  import java.awt.*;
8  import java.io.*;
9
10 public class Exam3_11 extends Canvas implements Serializable
11 {
12     private String s = "Hello";
13     private boolean smiling = true;
14     private int width = 60;
15     private int height = 50;
16
17     public Exam3_11() {
18         setSize(width, height);
19     }
20
21     public void paint(Graphics g) {
22         g.drawOval(5,5,50,30);
23         g.drawOval(12,12,15,8);
24         g.drawOval(30,12,15,8);
25         g.fillOval(19,12,8,8);
26         g.fillOval(37,12,8,8);
27         if (smiling)
28             g.drawArc(22,20,16,8,180,180);
29         else
30             g.drawArc(22,24,16,8,0,180);
31         g.drawString(s, 16, 45);
32     }
33
34     public void setText(String ss) {
35         s = ss;
36     }
37
38     public String getText() {
39         return s;
40     }
41
42     public void setSmiling(boolean b) {
43 }
```

```

41         smiling = b;
42     }
43
44     public boolean getSmiling() {
45         return smiling;
46     }
47
48     public void setWidth(int w) {
49         width = w;
50         setSize(width, height);
51     }
52
53     public int getWidth() {
54         return width;
55     }
56
57     public void setHeight(int h) {
58         height = h;
59         setSize(width, height);
60     }
61
62     public int getHeight() {
63         return height;
64     }
65 }
```

Lines 32 to 38 define the “text” property using a pair of set/get methods. So the user of the bean can now type in any text string to replace the original string “Hello” below the face. Lines 40 to 46 define the “smiling” property using a pair of set/get methods. Thus, when the user selects “False” for the property smiling, an angry face will be displayed. Otherwise, a smiling face is displayed by default. Lines 48 to 64 use two pairs of set/get methods to define two more properties: width and height of the Canvas. Figure 3.10 illustrates the bean Exam3_11 after it was loaded into the BeanBox.

QUICK-REVIEW QUESTIONS

1. The “smiling” property in Example 3.11 is a boolean property. However, the program used a *set/get* method pair instead of a *set/is* method pair as shown in the Code Pattern 3.4. How do you explain this?
2. In the Property window (the property sheet) for the bean Exam3_11, the “smiling” property item shows a drop-down list with “True” and “False” list items. How do we create a non-boolean property with a drop-down list of more than two items?
3. When I changed the value for the property “name,” nothing was changed to the bean in the BeanBox. What is the purpose for that property?

3.2.4.2 Removing Properties from a Bean Recall that bean Exam3_2 defined in Example 3.2 is inherited from JPanel and displayed 11 properties when it was loaded into the BeanBox. These 11 properties are debugGraphicsOptions, opaque, requestFocusEnabled, doubleBuffered, alignmentY, alignmentX, autoScrolls, foreground, background, font, and verifyInputWhenFocusTarget.

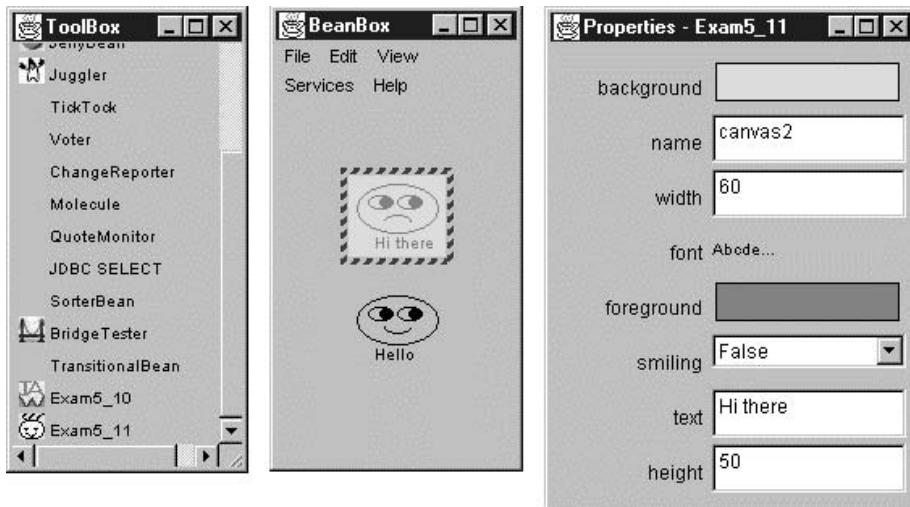


FIGURE 3.10. The property sheet for the bean Exam3_11.

If we are not interested in some of the properties, we can remove them from the property sheet by the BeanInfo class.

The example discussed below is based on Example 3.2 but uses a BeanInfo class to control the property sheet. We introduce one property called “text” but hide away 10 properties from the original property sheet.

Example 3.12 Manipulating the property sheet with a BeanInfo class:

```
//Exam3_12.java
//Based on Exam3_2, adding a property "text", and removing
several
//properties from the property sheet.

package Pack3_12;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class Exam3_12 extends JPanel implements Serializable {

    private JLabel l = new JLabel("A simple JLabel with raised
        border");
    private int width=200, height=100;

    public Exam3_12() {
        setSize(width, height);
        setBackground(Color.yellow);
        l.setFont(new Font("TimesNewRoman", Font.BOLD, 20));
        l.setForeground(Color.red);
    }
}
```

```

        l.setBorder(BorderFactory.createRaisedBevelBorder());
        add(l);
    }

    public void setText(String name) {
        l.setText(name);
    }

    public String getText() {
        return l.getText();
    }
}

```

The last two methods represent a pair of *set/get* methods for the new property “text” so that the user can modify the `JLabel` text at the design time. The BeanInfo class source file is given below:

```

1  //Exam3_12BeanInfo.java
2
3  package Pack3_12;
4
5  import java.beans.*;
6  import java.awt.*;
7
8  public class Exam3_12BeanInfo extends SimpleBeanInfo {
9      private final static Class beanClass = Exam3_12.class;
10     public PropertyDescriptor[] getPropertyDescriptors() {
11         try {
12             PropertyDescriptor background =
13                 new PropertyDescriptor( "Background",
beanClass);
14             PropertyDescriptor text =
15                 new PropertyDescriptor( "Text", beanClass);
16             background.setBound(true);
17             text.setBound(true);
18             PropertyDescriptor pv[] = {background, text};
19             return pv;
20         }
21         catch (IntrospectionException e)
22             { throw new Error(e.toString()); }
23     }
24 }
25
26     public Image getIcon(int iconKind) {
27         if (iconKind == BeanInfo.ICON_COLOR_16x16)
28             { Image img = loadImage("star.gif");
29              return img;
30             }
31         if (iconKind == BeanInfo.ICON_COLOR_32x32)
32             { Image img = loadImage("wja.gif"); }

```

```

33         return img;
34     }
35     return null;
36 }
37
38 public BeanDescriptor getBeanDescriptor() {
39     return new BeanDescriptor(beanClass);
40 }
41 }
```

Line 8

```
public class Exam3_12BeanInfo extends SimpleBeanInfo {
```

defines the class name as `Exam3_12BeanInfo` following the Code Pattern: “Bean-Name + BeanInfo.” The superclass `SimpleBeanInfo` implements the `BeanInfo` interface, providing the explicit information about the methods, properties, events, and so on, of the bean. The public method `getPropertyDescriptors()` in line 10

```
public PropertyDescriptor[] getPropertyDescriptors() {
```

returns an array of `PropertyDescriptor`s describing the editable properties supported by this bean. The class `java.beans.PropertyDescriptor` describes one property that a Java bean exports via a pair of set/get methods. This example uses its constructor in the following form:

```
public PropertyDescriptor(String propertyName,
                        Class beanClass)
                        throws IntrospectionException
```

The first parameter `propertyName` represents the programmatic name of the property. The second parameter `beanClass` is the `Class` object for the target bean. Since the constructor of `PropertyDescriptor` throws `IntrospectionException`, an exception that occurs during introspection, we have to put the `PropertyDescriptor` initialization part into a `try` block and catch `IntrospectionException` in the `catch` block.

Lines 16 and 17 set both *text* and *background* as bound property, which we will discuss shortly.

Line 38

```
public BeanDescriptor getBeanDescriptor() {
```

returns a `BeanDescriptor`, providing overall information about the bean. Line 39 uses a constructor of `BeanDescriptor` with a single parameter in the following form:

```
BeanDescriptor(Class beanClass),
```

which creates a `BeanDescriptor` for a bean that does not have a customizer.



FIGURE 3.11. The BDK window capture for loading Exam3_12.jar.

Loading the bean Exam3_12 into the BeanBox, you will see only two properties displayed in the property sheet, as shown in Figure 3.11.

QUICK-REVIEW QUESTIONS

1. What is a bound property? Why do we need bound properties?
2. Can we control those visible *events* and *methods* using the similar techniques discussed for *properties*?
3. What is a bean customizer?
4. How do we customize the property sheet with a customizer presented for a bean?

3.3 CONNECTION MODEL OF JAVABEANS

The connection model of JavaBeans allows us to combine components into applets, applications, or composite components. We discuss the connection model of JavaBeans with BDK 1.1 and Bean Builder 1.0.

3.3.1 Assembling Java Beans in BDK

The most interesting benefit of component-based software development is that we can build new software components from assembling existing software components. JavaBeans architecture provides a simple environment for component-based software development. We use one example to demonstrate how to assemble Java beans, that is, to create a new Java bean based on existing beans.

Example 3.13 Assembling Java beans in BDK: Suppose we want to use two `ExplicitButton` beans and one animated `Juggler` bean to create a new bean. We will label the buttons `Start` and `Stop`, which will start and stop the animation respectively.

Step 1: Start the BeanBox.

Step 2: Drop a `Juggler` bean and two `ExplicitButton` bean instances into the BeanBox.

Step 3: Select an `ExplicitButton` instance. On the properties sheet, change the label property to `Start`. Select a second `ExplicitButton` instance and change its label to `Stop`.

Step 4: Choose the `Start` button. Select the `Edit – Events – button push – actionPerformed` menu items in the specified order.

This causes a rubber band line to track between the `Start` button and the cursor. Click on the `Juggler` instance. This brings up the `EventTargetDialog` window. This list contains `Juggler` methods that take no arguments or arguments of type `actionPerformed`.

Step 5: Select the `startJuggling` method and click on `OK`. You will see a message that the `BeanBox` is generating adapter classes.

Step 6: Repeat steps 4 and 5 on the `Stop` button, but choose the `stopJuggling` method in the `EventTargetDialog`.

Clicking on the `Start` and `Stop` buttons will now start and stop the `Juggler`. Figure 3.12 shows a window capture of the result.

Below is a general description of what happened:

- The `Start` and `Stop` buttons are *event sources*. Event sources *fire events* at *event targets*. In this example, the `Juggler` bean is the event target.
- You choose the type of event that the event source will fire when you choose the `Start` button and choose an event method (via the `Edit – Event` menu item).
- You select the event target bean when you connect the rubber band line to another bean.
- The `EventTargetDialog` lists methods that can accept that type of event or that take no parameters. When you choose a method in the `EventTargetDialog`, you are specifying the method that will receive the fired event and act on it.

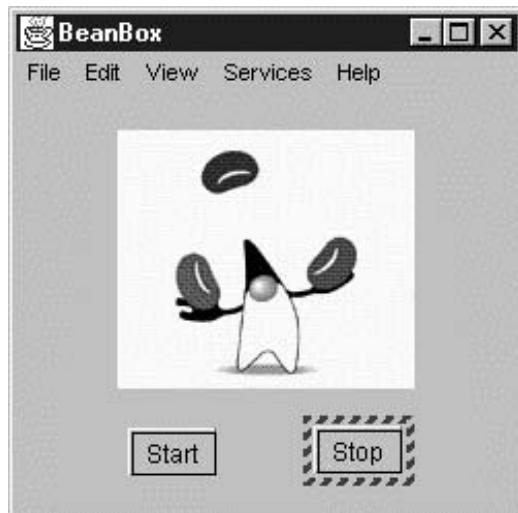


FIGURE 3.12. Assembling Juggler with two `ExplicitButtons`.

Step 7: Resize the BeanBox window to appropriate width and height and then select File – MakeApplet ... menu item. Select the right directory and JAR file name at the dialog window “Make an Applet.” This will generate an applet corresponding to a BeanBox layout. The data and classes needed for this applet are packaged into a JAR file. The applet itself is a bean and it can be read back into the BeanBox if desired.

QUICK-REVIEW QUESTIONS

1. After Step 7 above, I cannot display the generated applet using either a Web browser or appletviewer. What is the problem?
2. After the assembling procedure, we should have a new Java bean. What is the superclass for the resulting bean? Is there any relationship between the superclass of the result bean and those superclasses of assembled beans?
3. What properties will be displayed in the property sheet of the resulting bean? What visible events and methods will be displayed?

Note that the generated applet requires a Java 1.1-compliant Web browser to display. The appletviewer always works for this purpose. Alternatively, you can use the HotJava browser. You might need to modify the generated HTML file and add `</body></html>` to the end of the file.

3.3.2 Assembling Java Beans in Bean Builder

The Bean Builder 1.0 was released in January 2002 by Sun Microsystems, which allows the visual assembly of an application by instantiating and setting the properties and event handling methods of components in JavaBeans component infrastructure. The static structure of an application is specified by connecting lines with different colors. The dynamic behavior of the application is expressed by event handling between components. The state of the application is saved to and restored from an XML file.

The Bean Builder uses a modifiable GUI, called *palette*, to present reusable beans as illustrated in Figure 3.13. Note here that all Java Swing components are treated as beans. The content and format of the palette are determined by an XML file. The default palette file is named `palette.xml` located in the “lib” subdirectory of the Bean Builder installation.

In addition to this palette interface, Bean Builder 1.0 has a container window, called *designer*, for initiating, selecting, and assembling components. When a component is

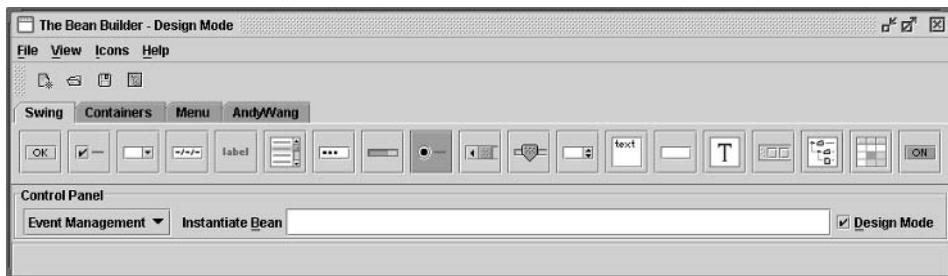


FIGURE 3.13. The Bean Builder palette.

selected from the palette and dropped into the designer, nine blue and white resize handles appear around the new component. The center handle allows the user to move the component around the designer. The handles on the edges and corners allow the user to resize the component in the direction indicated by the cursor, as shown in Figure 3.14 for the component JComboBox. The four dark gray connection handles on the outside of the component (as illustrated in Figure 3.14 for the component JLabel) act as anchors or ports for event handling or setting properties.

When a component is selected in the designer, all its public properties are shown in the Property Inspector window as shown for the JComboBox in Figure 3.15.

Can we drop a Java object not on the palette into the designer? The answer is “yes.” Let us instantiate an “invisible” component, DefaultComboBoxModel, and load it into our designer panel. First, put your cursor focus in the “Instantiate Bean” text field right below the palette in the top window. Then, enter the text in the text field

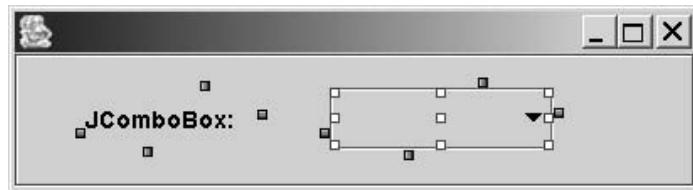


FIGURE 3.14. Two components dropped in the designer.

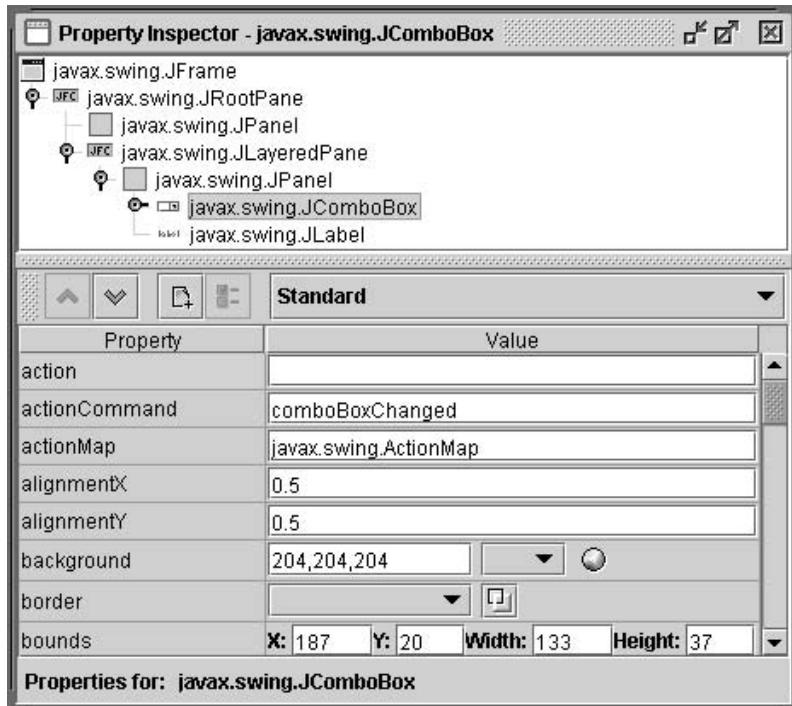


FIGURE 3.15. The property editor for the JComboBox component.

`javax.swing.DefaultComboBoxModel` and press the Enter key. You will see a visual proxy component appear in the upper left corner of your designer panel. Point your mouse cursor at the center handle of this proxy component and drag it to a new location, as shown in Figure 3.16.

The Bean Builder provides a graphical means of creating bean components by building the components from existing components in JAR files or by assembling components from the base JDK API packages. In order to allow composition of multiple components into a single application, Bean Builder provides two operations to assemble two or more bean components: (1) Set Property and (2) Event Adapter. We will discuss these two connecting operations below with examples.

1. Composition by Property Customization Property customization involves the manipulation of properties between two or more beans. Through property customization, a public property of a bean can be manipulated by another bean. This is done in Bean Builder through the “Set Property” operation.

Example 3.14 In Figure 3.16, there are three components. Let us try to connect the empty dropdown list component (`JComboBox`) to the `DefaultComboBoxModel` component using the “Set Property” operation provided by Bean Builder 1.0. The source component is `DefaultComboBoxModel` and the target component is the `JComboBox` component. The purpose is to set the property of the target component with the source component itself. Thus, the source component becomes a property of the target component.

Step 1: Click on any one of the four connection handles surrounding the source component, `DefaultComboBoxModel` instance, and drag the mouse over to a connection handle surrounding the `JComboBox` component.

Step 2: Release the mouse over the `JComboBox` connection handle. Notice that the connection line has red color now and a popup menu appears as shown in Figure 3.17.

Step 3: Select the *Set Property* radio button. The items in the list are target component methods whose parameter matches the type of the source component.

Step 4: Select the “model(ComboBoxModel)” list item and select the *Finish* button. The interaction wizard is dismissed and the `DefaultComboBoxModel` instance is used as the model for the `JComboBox` component. Notice that the color for the connection line now becomes green.

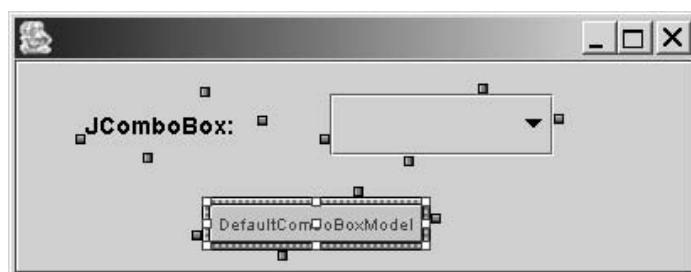


FIGURE 3.16. A proxy component.

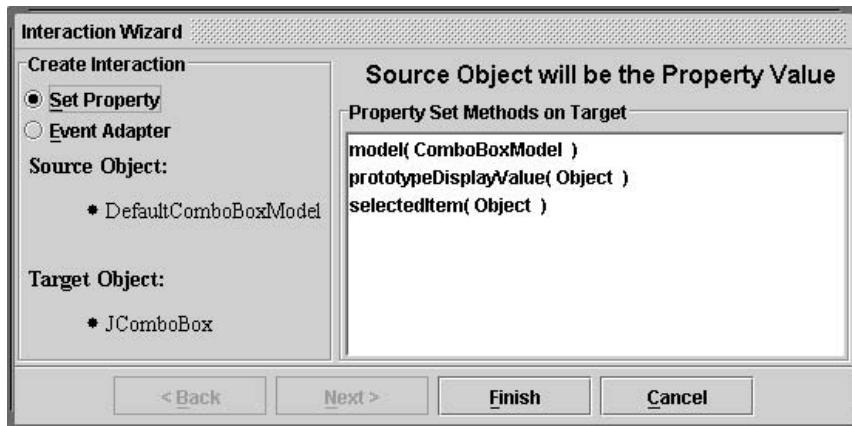


FIGURE 3.17. Set property popup window.

2. *Composition by Event Handling* Let us add two more text field components to the designer panel based on Figure 3.16. Our purpose is to connect JTextField-1, JTextField-2, and JComboBox such that when pressing Enter key in the text fields, the content of JTextField-1 will be added into the drop-down list, while the content of JTextField-2 will be selected in the drop-down list if the content matches one of the list items. The basic components are shown in Figure 3.18.

Step 1: Click on a connection handle for JTextField-1 and drag the cursor to a DefaultComboBoxModel connection handle.

Step 2: Select the Event Adapter radio button in the popup wizard, as shown in Figure 3.19.

Step 3: Select the “action” item in the “Event Sets” list and “actionPerformed(ActionEvent)” in the “Event Methods” list. Then press the *Next* button.

Step 4: Select the target method “addElement(Object)” and press the *Next* button.

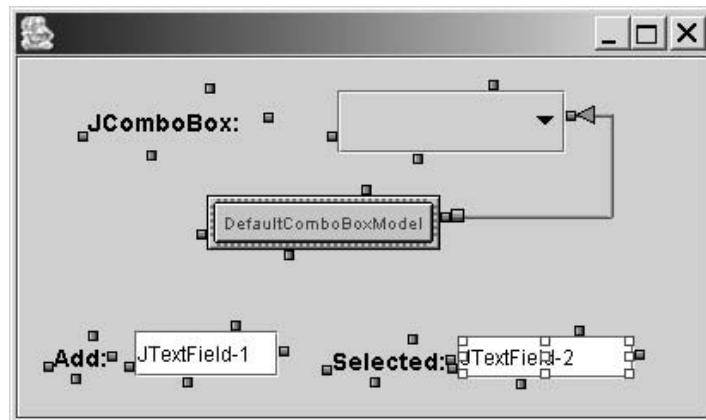


FIGURE 3.18. Basic components for Example 3.14.

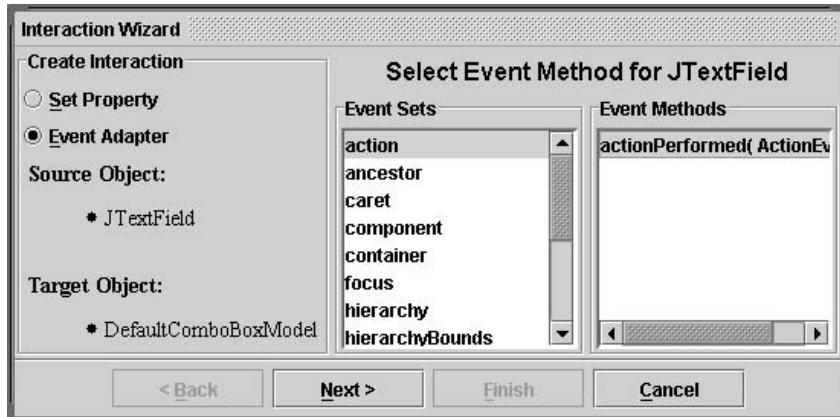


FIGURE 3.19. Event Adapter has been selected.

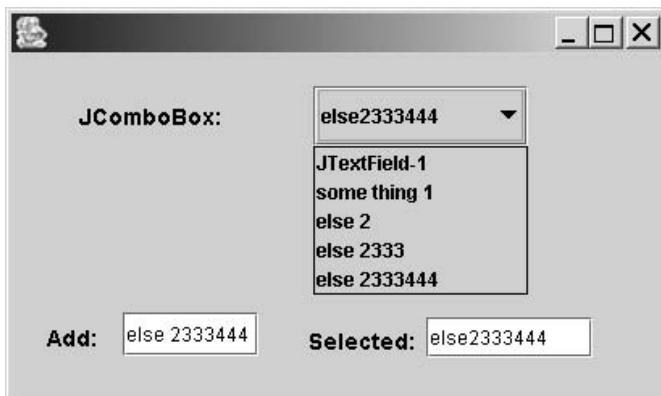


FIGURE 3.20. Testing the runtime application.

Step 5: Select the “`getText()`” item in the “Source Methods” list and select the *Finish* button.

A new event handler is created that adds the string in the JTextField-1 to the drop-down list as a result of the `actionPerformed` method when the Enter key is pressed.

Step 6: Create a new event adapter for the JTextField-2 using the interactive wizard. This connection selects the item in the drop-down list when the Enter key is pressed in the JTextField-2. This time the source component is JTextField-2, the Target component is DefaultComboBoxModel, the source method is `actionPerformed(ActionEvent)`, and the target method is `setSelectedItem(Object)`.

Step 7: Change the Bean Builder from *Design Mode* to *Runtime Mode* by unselecting the check box on the top window. In a popup window, the application behaves as it was designed. Type some string value in the JTextField-1 and press Enter key. The string will be added at the end of the drop-down list. Type some string value in the JTextField-2 and press Enter key. The string will become selected in the drop-down list, as shown in Figure 3.20.

3. *Working with User-Developed Beans* There are two ways to obtain an instantiation of a component outside of Java Swing packages:

Method 1: First, put your cursor focus in the “Instantiate Bean” text field right below the palette in the top window. Then, enter the complete path of your main class file for your component. We have followed this approach as we instantiate a proxy component for javax.swing.DefaultComboBoxModel in Figure 3.16.

Method 2: Edit the default palette.xml to include your components. The palette configuration file has a very simple structure, as demonstrated in the code list below. Figure 3.13 was created with the XML file shown in Figure 3.21.

```
<?xml version="1.0"?>
<!-- This is a palette configuration file for the Bean Builder. -->
<!DOCTYPE palette [
    <!ELEMENT palette (tab+)>
    <!ATTLIST tab name CDATA #REQUIRED>
    <!ELEMENT tab (item+)>
    <!ELEMENT item (#PCDATA)>
]>
<palette>
<tab name="Swing">
    <item>javax.swing.JButton</item>
    <item>javax.swing.JCheckBox</item>
    <item>javax.swing.JComboBox</item>
    <item>javax.swing.JFormattedTextField</item>
    <item>javax.swing.JLabel</item>
    <item>javax.swing.JList</item>
    <item>javax.swing.JPasswordField</item>
    <item>javax.swing.JProgressBar</item>
    <item>javax.swing.JRadioButton</item>
    <item>javax.swing.JScrollPane</item>
    <item>javax.swing.JSlider</item>
    <item>javax.swing.JSpinner</item>
    <item>javax.swing.JTextArea</item>
    <item>javax.swing.JTextField</item>
    <item>javax.swing.JTextPane</item>
    <item>javax.swing.JToolBar</item>
    <item>javax.swing.JTree</item>
    <item>javax.swing.JTable</item>
    <item>javax.swing.JToggleButton</item>
</tab>
<tab name="Containers">
    <item>javax.swing.JApplet</item>
    <item>javax.swing.JEditorPane</item>
    <item>javax.swing.JInternalFrame</item>
    <item>javax.swing.JDialog</item>
    <item>javax.swing.JFrame</item>
    <item>javax.swing.JOptionPane</item>
    <item>javax.swing.JPanel</item>
```

FIGURE 3.21. Palette configuration file.

```

<item>javax.swing.JTabbedPane</item>
<item>javax.swing.JFrame</item>
<item>javax.swing.JColorChooser</item>
<item>javax.swing.JFileChooser</item>
<item>javax.swing.JScrollPane</item>
</tab>
<tab name="Menu">
    <item>javax.swing.JMenu</item>
    <item>javax.swing.JMenuBar</item>
    <item>javax.swing.JMenuItem</item>
    <item>javax.swing.JSeparator</item>
    <item>javax.swing.JCheckBoxMenuItem</item>
    <item>javax.swing.JRadioButtonMenuItem</item>
    <item>javax.swing.JPopupMenu</item>
</tab>
<tab name="AndyWang">
    <item>sunw.demo.molecule.Molecule</item>
    <item>sunw.demo.sort.SortItem</item>
</tab>
</palette>

```

FIGURE 3.21. (continued)

Notice that at the end of the palette configuration file in Figure 3.21, one tab is added named as “AndyWang.” There are two components under this tab, which will be used in our discussion next.

3.4 DEPLOYMENT MODEL OF JAVABEANS

JavaBeans components are packaged and delivered in JAR files that contain the class file, serialized files, and the resources for the component along with a file called the *Manifest* that provides information about its contents. In BDK 1.1, components are deployed by moving them into a designated directory: BDK-Home/jars. These components will be displayed in the ToolBox frame of BDK when started. Components and applications built with BDK can be saved as a serialized file or a Java applet to be reused in a Web page.

Bean Builder 1.0 provides much flexible deployment method. In principle, you can have your component anywhere on your local computer, as long as your component JAR files are in the class path of the Java runtime system. The Java Swing components are deployed by default in Bean Builder 1.0. However, a user can load and display his/her components in the Bean Builder palette by modifying the palette.xml file. As an example, Figure 3.22 illustrates some user-designed beans loaded into the palette.

Note that some beans in Figure 3.22 do not have an icon on the palette. In order to show a customized icon for a bean when it is imported into the builder tool, the bean designer has to create a custom BeanInfo class for his/her bean with a public method `getIcon(int iconKind)` to return the icon. The predefined icon types are:

- `BeanInfo.ICON_MONO_16x16`
- `BeanInfo.ICON_MONO_32x32`

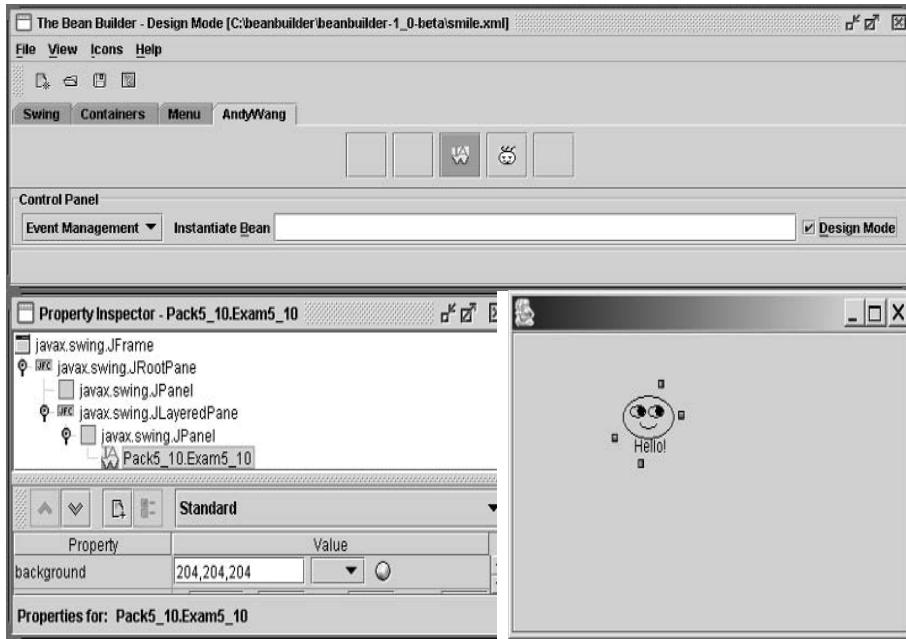


FIGURE 3.22. Load and display user-designed beans.

- BeanInfo.ICON_COLOR_16x16
- BeanInfo.ICON_COLOR_32x32

Figure 3.23 gives a sample BeanInfo class for the *smile* bean in Figure 3.22.

A user can also load their components into the designer panel by instantiating them in Bean Builder 1.0. After designing and connecting components into a new application, the user may save the application as an XML file, which is called the long-term persistence XML format that can be reloaded by a builder tool. This XML file retains all the design information for the application. Therefore, JavaBeans components can also be deployed with this XML file along with the JAR files. Figure 3.24 gives a sample code to reconstruct an XML design.

Example 3.15 In this example, we will construct an application, save it to an XML file, and reload and run it using the Java program in Figure 3.24. The reusable components are **JSlider.jar** and **JProgressBar.jar**. The **JSlider.jar** is the source component and **JProgressBar** is the target component. We will connect these two components so that when the user moves the slider, the progress bar will change its value correspondingly.

Step 1: Start Bean Builder 1.0. Select **JSlider.jar** and **JProgressBar.jar** from the Swing tab on the palette and drop them into the designer panel.

Step 2: From one connecting handle surrounding the source component, **JSlider.jar**, drag a line pointing to one of the handles surrounding the target component, **JProgressBar.jar**.

```
//Exam5_11BeanInfo.java
//The BeanInfo class for the bean inheriting from Canvas

package Pack5_11;

import java.beans.*;
import java.awt.*;

public class Exam5_11BeanInfo extends SimpleBeanInfo {

    public Image getIcon(int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_16x16) {
            Image img = loadImage("smile.gif");
            return img;
        }

        if (iconKind == BeanInfo.ICON_COLOR_32x32) {
            Image img = loadImage("smile.gif");
            return img;
        }
        return null;
    }
}
```

FIGURE 3.23. A sample BeanInfo class.

```
import java.io.*;
import java.beans.XMLDecoder;

public class Test {
    public static void main(String arg[]) {
        try {
            InputStream is = new BufferedInputStream(
                new FileInputStream(arg[0]));
            XMLDecoder d = new XMLDecoder(is);
            Object o = d.readObject();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Usage: java Test
                XmlFileName.xml");
        }
    }
}
```

FIGURE 3.24. A sample Java code to reconstruct a design.

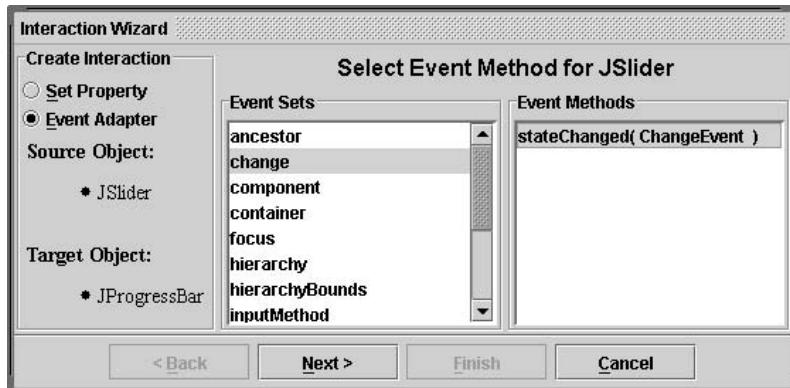


FIGURE 3.25. Select source event method.

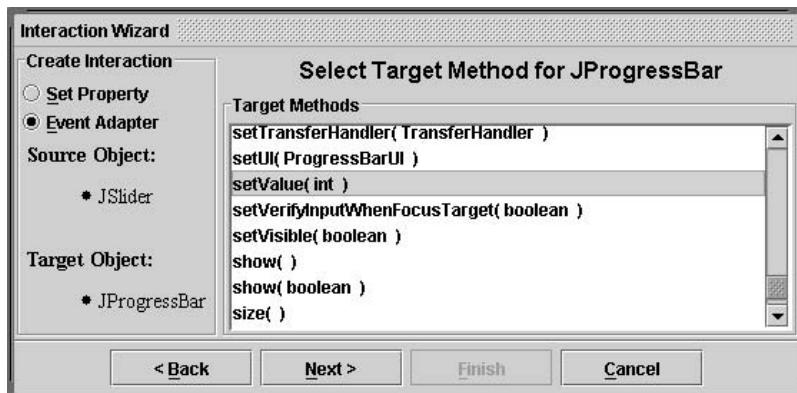


FIGURE 3.26. Select target method.

Step 3: Select `stateChanged(ChangeEvent)` as the event method for `JSlider`, as shown in Figure 3.25.

Step 4: Select `setValue(int)` as the target method for `JProgressBar`, as shown in Figure 3.26.

Step 5: Select `getValue()`, whose return value will be the integer value passed to the target method as parameter (Figure 3.27).

Step 6: Press the “Finish” button and the result in the designer panel will look like Figure 3.28.

Step 7: Save the design into an XML file called `SliderBar.xml`.

Step 8: Compile the source code in Figure 3.24 and issue the following command from a command shell window:

```
java Test SliderBar.xml
```

Step 9: Try to move the slider, the progress bar should display the corresponding value of the slider, as shown in Figure 3.29.

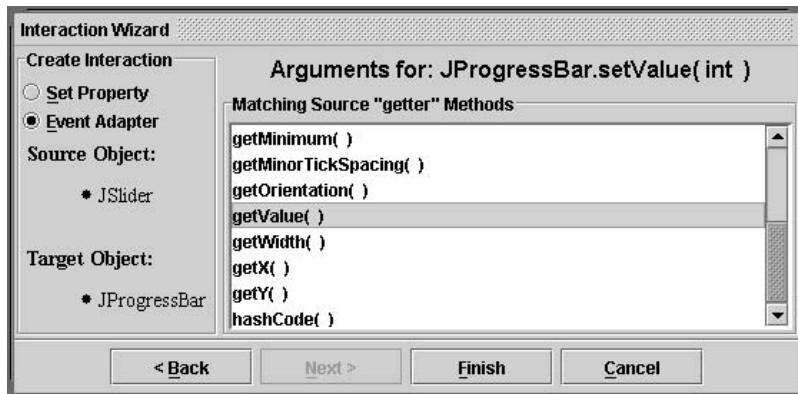


FIGURE 3.27. Select the argument for the target method.

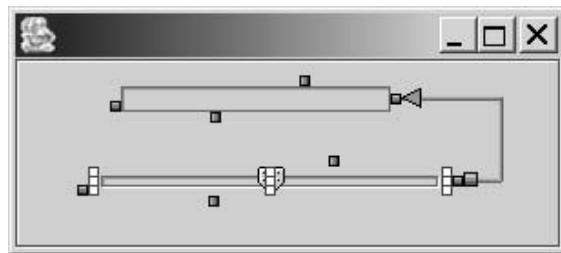


FIGURE 3.28. The connection of the two components.

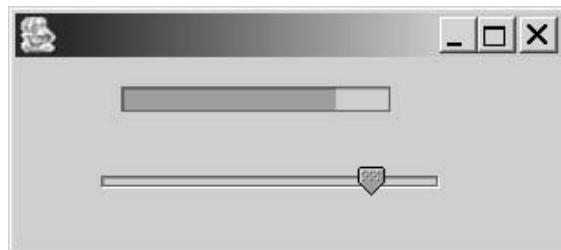


FIGURE 3.29. Runtime result of Example 3.15.

3.5 EXAMPLES AND LAB PRACTICE

Lab 3.1 Install Java Beans Development Kit (BDK 1.1) on Your Computer

In order to develop Java beans, you need to install the *JavaBeans Development Kit (BDK)* first on your computer. The BDK can be downloaded free of charge from the Sun Microsystems website: <http://java.sun.com/beans/software/index.html>. At the time of writing this book (July of 2000), the latest version of BDK was BDK 1.1 released in April 1999.

Before downloading BDK, make sure that you have installed the Java 2 Standard Edition SDK version 1.3 released in May 2000, since BDK 1.1 depends on the Java

2 platform. BDK is available for Windows, Solaris, and Linux. The description below is assuming that you are using a Windows computer.

Step 1: Download the BDK installer software. You can download it from the Sun Microsystems web-site. The name and size of the BDK installer are:

bdk1_1-win.exe	2,510,420 bytes
----------------	-----------------

Step 2: Run the BDK installer bdk1_1-win.exe.

After running the BDK installer, you will be prompted to where you want to install it. The BDK should be installed as a root directory in a drive. For convenience of discussion below, I suppose you installed the BDK at the following directory:

D:\BDK1.1

Of course, you can install it on another disk drive.

Step 3: Check the files and directories.

After installing the BDK software, check the installation directory D:\BDK1.1, which should have the following files and directories:

D:\BDK1.1>dir

```

Volume in drive D has no label
Volume Serial Number is 07CE-0B11
Directory of D:\BDK1.1

.
..
UNINST~1 <DIR>      09-13-99  9:36a UninstallerData
LIB       <DIR>      09-13-99  9:37a lib
JARS      <DIR>      09-13-99  9:37a jars
INFOBUS   JAR      76,262 09-13-99  9:37a infobus.jar
DOC       <DIR>      09-13-99  9:37a doc
DEMO      <DIR>      09-13-99  9:37a demo
BEANBOX   <DIR>      09-13-99  9:37a beanbox
MAKEFILE   564        09-13-99  9:37a Makefile
LICENS~1  HTM      5,968   09-13-99  9:37a LICENSE.html
GNUMAK~1   452        09-13-99  9:37a GNUMakefile
BEANS     GIF      7,012   09-13-99  9:37a beans.gif
README~1  HTM      4,559   09-13-99  9:37a README.html
               6 file(s)  94,817 bytes
               8 dir(s)  6,258.49 MB free

```

Step 4: Update the PATH variable. If you are using Windows 95/98, edit the file C:\AUTOEXEC.BAT using a DOS editor or using the system editor (Choose “Start,” “Run” and enter sysedit, then click OK. The system editor starts up with several windows showing. Go to the window that is displaying AUTOEXEC.BAT.). Look for the PATH statement and update it. (If you do not have one, add one.) For example, in the following PATH statement, we have added the BeanBox directory at the right end:

PATH C:\WINDOWS;D:\JDK1.3\BIN;D:\BDK1.1\beanbox

To make the path take effect in the current command prompt window, execute the following:

```
C:>C:\autoexec.bat
```

If you are using Windows NT/2000/XP, change your path accordingly.

Step 5: Start the BDK environment. At a command line of any DOS window, type “run” then hit the Enter key. You will have four windows popped up.

- The left-hand window is the ToolBox palette displaying available beans that can be used to build new beans.
- The middle window is the main BeanBox composition window where new beans are assembled.
- The upper right-hand window is a Property window showing the properties for the currently selected bean.
- The lower right-hand window is a Method Tracer displaying BeanContext services.

Lab 3.2 Get Familiar with BDK 1.1

This lab practice gets yourself familiar with the BDK environment by constructing an application using existing beans in the ToolBox and BDK.

Step 1: From a command shell, change to the BDK installation directory and start it. You should see the interface as shown in Figure 3.30.

Let us get familiar with BeanBox menus (File, Edit, and View menus) before we move on.

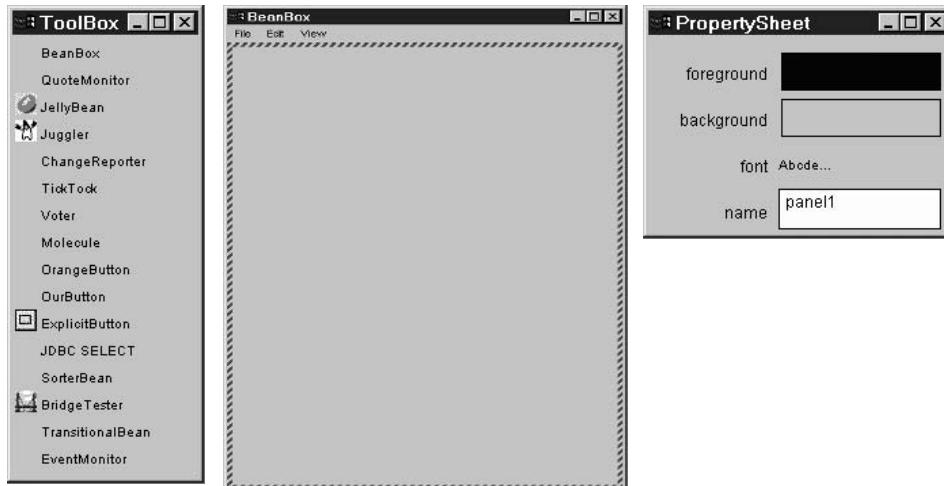


FIGURE 3.30. The BDK interface.

1. File Menu

Menu Item	Action
Save	Saves the beans currently in the BeanBox, including each Bean's size, position, and internal state. The saved file can be loaded via <code>File -- Load</code> .
SerializeComponent...	Saves the beans in the BeanBox to a serialized (<code>.ser</code>) file. This file must be put in a <code>.jar</code> file to be usable by the BeanBox.
MakeApplet...	Generates an applet from the BeanBox contents.
Load...	Loads saved files into the BeanBox. This command will not load <code>.ser</code> files.
LoadJar...	Loads a JAR file's contents into the ToolBox.
Print	Prints an image of the BeanBox contents.
Clear	Removes the BeanBox contents.
Exit	Quits the BeanBox <i>without offering to save</i> .

2. Edit Menu

Menu Item	Action
Cut	Removes the bean selected in the BeanBox. The cut bean is serialized and can then be pasted.
Copy	Copies the bean selected in the BeanBox. The copied bean is serialized and can then be pasted.
Paste	Drops the last cut or copied bean into the BeanBox.
Report...	Generates an introspection report for the selected bean.
Events	Lists the selected bean's event-firing method, grouped by the Java interface that declares the methods.
Bind property...	Lists all of the selected bean's bound property methods, if any.

3. View Menu

Menu Item	Action
Disable Design Mode/Enable Design Mode	Removes the ToolBox and the Properties sheet from the screen. Eliminates all BeanBox design and test behavior (selected bean, and so on) and makes the BeanBox behave like an application.
Hide Invisible Beans>Show Invisible Beans	Hides or shows beans with no GUI.

Step 2: From the left frame called ToolBox, drag a Juggler and two JellyBean beans into the BeanBox. This can be done with your mouse: first select the bean and click in the BeanBox frame.

Step 3: Change colors for the two JellyBean beans. Select one JellyBean with your mouse, you will see a dashed-angled line around it. The Property Sheet frame on the right-hand side lists all the properties for the current bean. The default color for the JellyBean is yellow as shown on the left side of Figure 3.31. Select the color field in the property sheet and change the color into green, as shown on the right side of Figure 3.31. Change the label to Start. Similarly, change the color of another JellyBean into red.

Step 4: Select the red button. Select the “Edit” menu on the BeanBox menu bar. Select the “Events” submenu. Select the “mouse” submenu. Select the “mouseClicked” menu item. Then attach the red line to the Juggler bean by selecting it. When you see a popup window called “EventTargetDialog,” select the startJuggling method from the list, as shown in Figure 3.32. Select the OK button.

Step 5: Conduct the similar actions for the red JellyBean button. This time, of course, select the stopJuggling method from the method list of the popup “EventTargetDialog” window.

Step 6: Now click the JellyBean buttons with your mouse and you will have a fully functional program with an interface as shown in Figure 3.33.

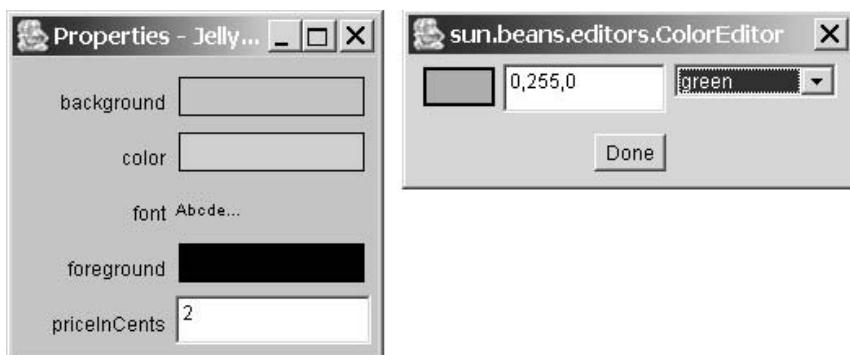


FIGURE 3.31. Change color for the JellyBean.

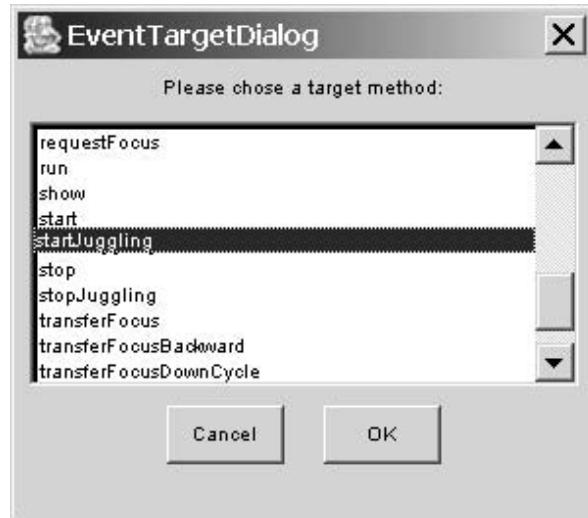


FIGURE 3.32. Select a mouse event handler.



FIGURE 3.33. A fully functional program built from beans.

Step 7: Now save your work by selecting “Save . . .” under the “File” menu and save the file as “testing.ser.” The .ser extension stands for serialized. It is not a required extension but we use it here just for reminding us of the file property.

You can also save the result into a Java applet so that you could upload it to a Web page. This is done by selecting “MakeApplet . . .” under the “File” menu. Let us also name our work as an applet named MyApplet, as shown in Figure 3.34.



FIGURE 3.34. Create a Java applet from BeanBox.

Step 8: Exit the BeanBox and restart it. Reload the saved serialized file “testing.ser,” and you should see the animation as shown in Figure 3.33.

To test your applet created in the last step, change to the “tmp/myApplet” subdirectory under “BDK1.1/beanbox” and launch appletviewer with the HTML file named `myApplet.html`. You should see a running Juggler in a Web page.

Lab 3.3 Create a Bean from a Java Application in BDK 1.1

Step 1: Find a Java application (not applet) from your previous program assignments.

Step 2: Convert the Java application into a bean following the 8 steps discussed in Section 3.2.

Step 3: After loading your bean into the ToolBox, select your bean by clicking the left button of your mouse on your bean name in the ToolBox window. Your mouse cursor should now change to a crosshair shape. Position the mouse cursor inside the BeanBox window and click your left button to place your bean in the BeanBox.

Step 4: Now you should see your bean with a dashed border in the BeanBox. The dashed border highlights the fact that the bean was selected. If you did not see such a selecting border around your bean, click your bean to select it. (For some beans, you need to click just outside the bean’s boundary to select it.)

Step 5: Check the Property window on the right-hand side and answer the questions below:

QUICK-REVIEW QUESTIONS

1. How many properties do you have?
2. What are they?
3. Where do they come from?

Lab 3.4 Create a Bean from a Java Applet in BDK 1.1

Step 1: Find a Java applet (AWT Applet or Swing JApplet, not application) from your previous program assignments.

Step 2: Convert the Java applet into a bean following the 8 steps discussed in Section 3.2.

Step 3: After loading your bean into the ToolBox, select your bean by clicking the left button of your mouse on your bean name in the ToolBox window. Your mouse

cursor should now change to a crosshair shape. Position the mouse cursor inside the BeanBox window and click your left button to place your bean in the BeanBox.

Step 4: Now you should see your bean with a dashed border in the BeanBox. The dashed border highlights the fact that the bean was selected. If you did not see such a selecting border around your bean, click your bean to select it. (For some beans, you need to click just outside the bean's boundary to select it.)

Step 5: Check the Property window on the right-hand side and answer the questions below:

1. How many properties do you have?
2. What are they?
3. Where do they come from?

Lab 3.5 Install Bean Builder 1.0 on Your Computer

Step 1: Download the Bean Builder 1.0 from Sun Microsystems [Sun2 2003]. The distribution file is called `builder.zip`.

Step 2: Create a new directory for the installation of Bean Builder 1.0. This directory will be referred to as the “builder root.”

Step 3: Unzip the `builder.zip` file in the builder root directory.

Step 4: Set your `JAVA_HOME` environment variable to point to the root of your J2SDK 1.4.0 or later distribution.

Step 5: Execute the `run.bat` batch file.

Lab 3.6 Component Connection in Bean Builder 1.0

Step 1: Start Bean Builder 1.0 and load one instance of the `molecule.jar` from BDK 1.1 into the designer panel.

Step 2: Drop into the designer panel two `JButton` instances. Rename the button text to “RotateX” and “RotateY” respectively.

Step 3: Connect source component “RotateX” to `molecule.jar`: Source event method: `actionPerformed(ActionEvent)`; target method: `rotateOnX()`.

Step 4: Connect source component “RotateY” to `molecule.jar`: Source event method: `actionPerformed(ActionEvent)`; target method: `rotateOnY()`.

The connection design should look like Figure 3.35.

Step 5: Change the mode of Bean Builder by unselecting the checkbox “Design Mode” in the Control Panel near the “Instantiate Bean” text field.

Step 6: In the newly popped up window, try to click “RotateX” and “RotateY” and observe the result.

3.6 SUMMARY

The JavaBeans specification defines a component infrastructure to build, customize, assemble, and deploy Java software components. It brings the advantages of component-oriented programming to the ever-growing Java development platform [Camp 1999]. As

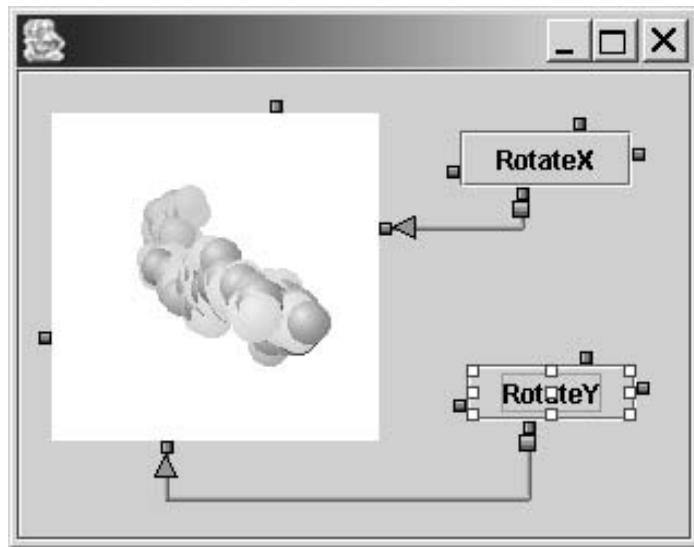


FIGURE 3.35. The connection of three components.

a component infrastructure, JavaBeans provides most of the requirements specified by the component theory discussed in Chapter 2. It provides properties, which are exposed through its interface by getter and setter methods. It provides methods, which are used to provide services to be called from other components. It provides an event handling model, which is critical for communications among beans. Finally, the BeanInfo interface describes what properties, methods, and events are exposed by a bean component. In addition, JavaBeans provides support for introspection and customization, which allow automatic analysis of beans and easy configuration of bean components.

One of the distinguishing features of JavaBeans is that components are visually composed into applications in a builder tool. Two builder tools have been discussed in this chapter: BDK 1.1 and Bean Builder 1.0, both from Sun Microsystems, Inc. BDK 1.1 has a BeanBox window for visually manipulating components. The composition method in BDK 1.1 is event handling. The Bean Builder 1.0 demonstrates new technologies in Java platform that allow the construction of applications using component assembly mechanisms. The Bean Builder 1.0 extends the capabilities of the BDK 1.1 by demonstrating new techniques for long-term persistence, layout editing, and dynamic event adapter generation, using the new dynamic proxy API that was introduced in Java 2 platform.

Bean Builder 1.0 comes with two main connecting operators for assembling components: “Set Property” and “Event Adapter.” The “Set Property” operator provides the designers with a technique of changing some properties of targeting components. The “Event Adapter” is used to connect an event from one component up to a method of another component. In other words, this operator provides for the “multiplication” composition in device beans discussed in Chapter 2. It is possible to create a chain of events to alter multiple components from one fired event. So the first component in the sequence would fire an event to the second component, which in turn would trigger the third component to react, and so on. The strength of Bean Builder with regard to the

component model and the connection model in the JavaBeans component infrastructure is mainly the visualization of the connections that can be made between components. The use of XML file to describe component design makes long-term persistence possible. XML is becoming a universal language for its independence between builder tools and computer architectures. However, neither BDK 1.1 nor Bean Builder 1.0 is a mature component technology yet. Their functionalities and usability have to be improved significantly before they could be a major market hit in component-oriented programming.

3.7 SELF-REVIEW QUESTIONS

1. The JavaBeans components are
 - a. known as “beans”
 - b. reusable software units written in Java
 - c. visually composed into composite components using a visual builder tool
 - d. all of the above
2. The JavaBeans components expose their _____ to builder tools for visual manipulation.
 - a. interfaces and implementation of the interfaces
 - b. the source code plus supporting classes
 - c. properties, public methods, and events
 - d. JavaDoc information and manifest file contents
3. BDK 1.1 stores reusable components in
 - a. ToolBox
 - b. BeanBox
 - c. Palette
 - d. Property sheet
4. Bean Builder 1.0 stores reusable components in
 - a. ToolBox
 - b. BeanBox
 - c. Palette
 - d. Property sheet
5. The *properties* of a bean refers to
 - a. the mechanism to communicate with other beans
 - b. the attributes, appearance, and behavior characteristics of the bean
 - c. the BeanInfo interface or the bean customizer class
 - d. the services this bean will provide for other beans
6. The *methods* of a bean refers to
 - a. the mechanism to communicate with other beans
 - b. the attributes, appearance, and behavior characteristics of the bean
 - c. the BeanInfo interface or the bean customizer class
 - d. the services this bean will provide for other beans

7. The *events* of a bean refers to
 - a. the mechanism to communicate with other beans
 - b. the attributes, appearance, and behavior characteristics of the bean
 - c. the BeanInfo interface or the bean customizer class
 - d. the services this bean will provide for other beans
8. In JavaBeans component infrastructure, a _____ fires an event, while a _____ provides a handler to response to the event.
 - a. listener bean, target bean
 - b. listener bean, source bean
 - c. source bean, target bean
 - d. target bean, source bean
9. Persistence enables a bean to
 - a. modify its attributes and appearance
 - b. save and restore its states
 - c. call other beans for services
 - d. introspect properties, methods, and events described in a BeanInfo class
10. Every JavaBean component has a visual interface.
 - a. True
 - b. False

Keys to Self-Review Questions

1. d 2. c 3. a 4. c 5. b 6. d 7. a 8. c 9. b 10. b

3.8 EXERCISES

1. Create a JavaBean component that displays the current time and date.
2. Find a Java program from your code library and change it into a JavaBean component.
3. Investigate the problems and their solutions for the following situation: When you load a JAR file into the Bean Builder 1.0, you get a null pointer error.
4. Investigate the problems and their solutions for the following situation: When you load a JAR file into the Bean Builder 1.0, you get “java.lang.ClassFormatError: Duplicate name” error.
5. How do you get your JavaBeans component to display a customized icon when it is imported into a Bean Builder tool?
6. How many different ways do you instantiate a bean?
7. How many different composition methods (operators) does BDK 1.1 provide? What are they?

8. For each operator you identified in the previous question about BDK 1.1, provide at least one example to show how this operator works.
9. How many different composition methods (operators) does Bean Builder 1.0 provide? What are they?
10. For each operator you identified in the previous question about Bean Builder 1.0, provide at least one example to show how this operator works.
11. How do you specify static relationship among JavaBean components in BDK 1.1 and Bean Builder 1.0?
12. How do you specify dynamic behaviors and dependency among JavaBean components in BDK 1.1 and Bean Builder 1.0?
13. What are the strength and limitations of JavaBeans?

REFERENCES

- [Camp 1999] Campione, M., Walrath, K., and Huml, A. *The Java Tutorial, Continued*, Addison-Wesley, 1999.
- [Sun 1997] Sun Microsystems. *JavaBeans Specification v1.0.1*, July 1997, <http://java.sun.com/products/javabeans/docs/spec.html>.
- [Sun1 2001] MageLang Institute, JavaBeans Short Course, and Stearns, B. JavaBeans 101, Sun Microsystems, Inc. <http://java.sun.com/products/javabeans/learning/tutorial/index.html>, 2001.
- [Sun2 2003] Davidson, M. The Bean Builder Tutorial, Sun Microsystems, Inc., <https://bean-builder.dev.java.net/guide/tutorial.html>, 2003.

4

ENTERPRISE JAVABEANS COMPONENTS

Objectives of This Chapter

- Introduce J2EE framework and EJB architecture
- Introduce the concepts of EJB component and its runtime environment
- Discuss the types of EJB components, connection, and their deployments
- Introduce the new features of EJB 2.x
- Distinguish between synchronous and asynchronous method invocations
- Provide step-by-step tutorials on building, deploying, and using EJB components

4.1 EJB ARCHITECTURE

4.1.1 Overview of the EJB Architecture and J2EE Platform

Sun Microsystems Inc. announced its Enterprise Java Bean (EJB) specification in 1998. The new specification of EJB 2.1 was released in 2002. The EJB 2.x architecture is a component architecture for development and deployment of component-based distributed applications. An EJB component is a reusable, WORA (Write Once Run Anywhere), portable, scalable, and compiled software component that can be deployed on any EJB servers such as Java 2 Platform Enterprise Edition (J2EE), JBoss, and WebLogic Enterprise environment. The EJB technology is part of J2EE, which provides a set of APIs, and other system services. The EJB implementations concentrate on business logic. J2EE is designed to support applications that provide enterprise services. J2EE not only supports EJB components but also supports Web components such as JSP and Servlets.

The EJB architecture makes enterprise application development much easier because there is no need to care for system level services such as transaction management, security management, multithreading management, and other common management issues.

The EJB architecture also supports WORA and portable solution. An EJB component can be developed once and then reused in many applications and deployed on many different platforms without recompilation or modification of the source code of the EJB component.

The EJB architecture also manages the EJB component life cycle from the creation to the termination including activation and deactivation of an EJB component.

The EJB architecture provides Web services and compatibility with CORBA.

An EJB component is a server-side component that provides services to remote or local clients, while a Java bean is a client-side component that is installed and run at the client side most of time. We can even have a Java bean running on the server side but it is hard to provide services to any remote clients. An EJB component is hosted by its container and the container is supported by J2EE or any J2EE-compliant tools.

4.1.2 J2EE Server

A J2EE compliant application server provides the following services:

- Java Naming and Directory Interface (JNDI) API, which allows clients to look up and locate the container where EJB components are registered and hosted.
- J2EE supports authentication and authorization for security purposes.
- J2EE supports HTTP service and EJB-related services, device access services, and multithreading services.

Figure 4.1 depicts the EJB infrastructure.

4.1.3 EJB Container

The EJB instances are running within the EJB container. The container is a runtime environment (set of classes generated by deployment) that controls an EJB component instance and provides all necessary management services for its whole lifetime. Below

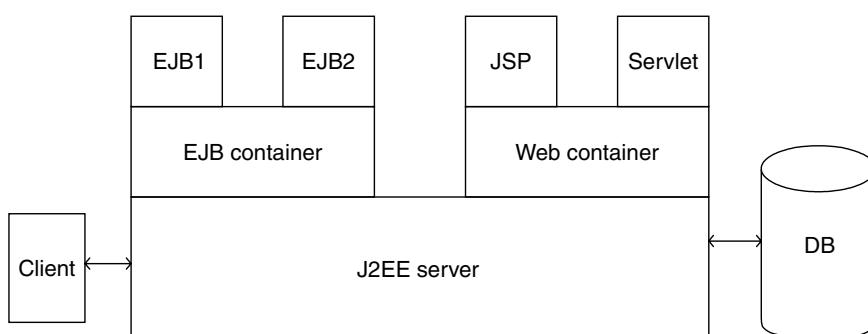
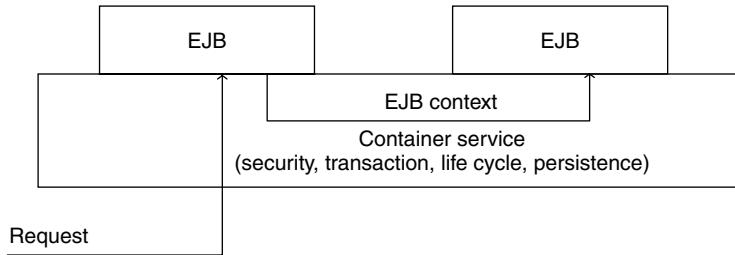


FIGURE 4.1. J2EE architecture.

**FIGURE 4.2.** EJB container.

is a list of such services [Sun1 2002]:

- Transaction management: ensuring transaction properties of multiple distributed transaction executions.
- Persistence management: ensuring a persistent state of an entity bean, which is backed up by database.
- Life cycle management: ensuring the EJB component state transitions in its life cycle.

The EJB container provides an interface for the EJB component to the outside world. All access requests to the EJB component and responses from the EJB component must get through the EJB container. The EJB container isolates EJB component from direct access by its clients. The container will intercept the invocation from clients to ensure the persistence, properties of transaction, and security of client operations on EJB. Figure 4.2 shows that the EJB container supports EJB components and an EJB component needs the container to reach outside and to obtain necessary information from its context interface.

The EJB container is in charge of generating an EJB home object, which helps to locate, create, and remove the EJB component object.

The EJB context interface provided by the EJB container encapsulates relevant information about the container environment such as the identity of an EJB component client, the status of the transaction, the remote reference to EJB itself, and so on.

4.1.4 EJB Component

An enterprise bean is a distributed component that lives in an EJB container and is accessed by remote clients over network via its remote interface or is accessed by other enterprise beans on the same server via its local interface. The EJB component is a remotely executable component deployed on its server and it is self-descriptive component specified by its Deployment Descriptor (DD) in XML format.

4.2 COMPONENT MODEL OF EJB

4.2.1 Overview

Each EJB component has a business logic interface exposed by the component so that clients can access the business logic operations via this interface without knowing the

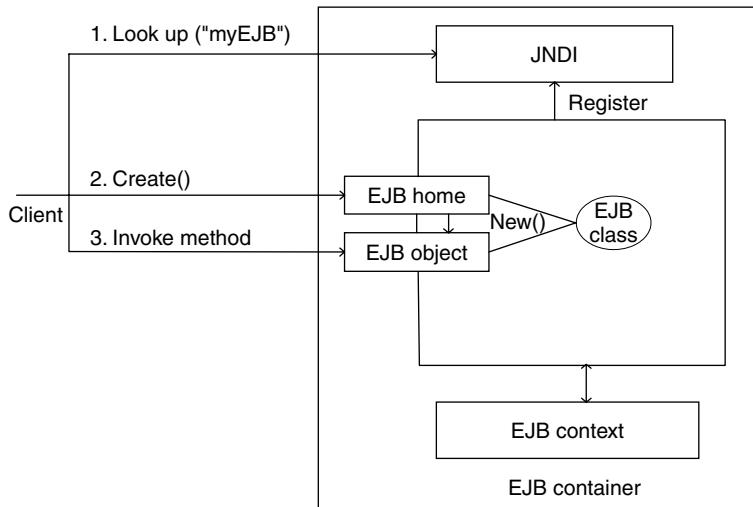


FIGURE 4.3. The interaction between a client and an EJB component.

detail implementation behind the interface. We call such interface as a *remote interface*. An instance of an EJB component is created and managed via its *home interface* by the EJB container. Every enterprise bean must have a home interface and a remote interface. The EJB component can be configured at the deployment time by specifying its DD. Figure 4.3 depicts the structure of an EJB component and the process flowchart of an interaction between a client and an EJB component.

The EJB class behind home and remote interfaces are designed to implement these two interfaces. An EJB component is a black-box component. A client of an EJB component only knows what the component does but not how it does. A client makes a request to an EJB component with its deployed name by looking up at JNDI to get the Object Reference (OR) of this EJB component. The client can then create an instance of this EJB component on the server according to the OR. Finally, the client invokes the interface methods of this EJB instance. Of course, an EJB must register itself with JNDI for clients to look up.

Enterprise beans are software components that can be embedded in various applications without recompiling or modifying their source code. They can be deployed in any EJB-compliant servers.

The EJB component model supports the following enterprise bean types:

- Stateless session beans that implement various business logics, such as language translation, logon process, tax calculation, and currency conversion.
- Stateless session beans wrapped in a Web service. Any existing enterprise bean can be encapsulated in an external Web service by a WSDL document describing the Web service endpoint that the bean implements. Such special beans for Web services do not provide interfaces that a regular EJB component provides.
- Stateful session beans, which play the same roles as stateless session beans except that they keep track of the states of the conversation from clients to the EJB components. For instance, a shopping cart bean can be a typical stateful session bean.

Any session bean, no matter a stateful or stateless one, does not support the persistence requirement for an entity bean in EJB component infrastructure:

- Message-driven beans, which were introduced lately, represent a new EJB component type that work in an asynchronous communication mode just like an event-driven event delegation model in Java.
- Bean Managed Persistence (BMP) entity beans, which are entity beans while their persistent storage management are taken care by themselves.
- Container Managed Persistence (CMP) entity beans, which are entity beans while their persistent storage management is specified by the deployment tool and managed by the container. However, the CMP entity beans do not need to handle any database SQL access. An entity bean is backed up by a relational database.

The remote interface of an EJB component implements `javax.ejb.EJBObject` interface, which in turn implements `java.rmi.Remote` interface. The home interface of an EJB component implements `javax.ejb.EJBHome` interface, which again implements `java.rmi.remote` interface. A local interface implements `javax.ejb.EJBLocalObject` interface and a local home interface implements `javax.ejb.EJBLocalHome` interface. The local interface is used by another EJB component running at the same server to access it so that it can reduce the overhead caused by remote access. The remote interface provides the location independence but it is more expensive. The local interface makes invocation more efficient. Another important difference between local and remote interfaces is that method invocation in local interface uses *passing by reference* and the method invocation in remote interface uses *passing by value*, which needs serialization, that is, marshaling and unmarshaling.

4.2.2 Session Beans

As its name implies, a session bean is an interactive bean and its lifetime is during the session with a specific client. A session bean corresponds to a particular client on the server. It responds to the behavior of a client and terminates when the client session is over. Session beans are often designed for business logic and flow control in front of entity beans. A session bean may make requests to another session bean or to other Web components such as JSP, Servlet, or HTML pages.

There are two types of session beans: stateless session beans and stateful session beans.

The following code shows an example of a stateless session bean that converts the temperature between Fahrenheit and Centigrade:

```
//Converter.java specifies the remote interface for this converter
//session bean component.

package converter;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.*;

public interface Converter extends EJBObject {
    public double cToF(double c) throws RemoteException;
```

```
public double fToC(double f) throws RemoteException;  
}  
  
//The file ConverterHome.java specifies the home interface for  
//this EJB component.  
  
package converter;  
import java.io.Serializable;  
import java.rmi.RemoteException;  
import javax.ejb.CreateException;  
import javax.ejb.EJBHome;  
  
public interface ConverterHome extends EJBHome {  
  
    Converter create() throws RemoteException, CreateException;  
}  
  
//The file ConverterBean.java specifies the EJB implementation  
//class for above interfaces of this component.  
  
package converter;  
import java.rmi.RemoteException;  
import javax.ejb.SessionBean;  
import javax.ejb.SessionContext;  
import java.math.*;  
  
public class ConverterBean implements SessionBean {  
  
    public double cToF(double c) {  
  
        double temp=0; temp=c*9/5+32;  
        return temp;  
    }  
  
    public double fToC(double f) {  
  
        double temp=0;      temp=(f-32)*5/9;  
        return temp;  
    }  
  
    public ConverterBean() {}  
    public void ejbCreate() {}  
    public void ejbRemove() {}  
    public void ejbActivate() {}  
    public void ejbPassivate() {}  
    public void setSessionContext(SessionContext sc) {}  
}  
// ConverterBean
```

A stateful session bean represents a specific client and holds the data for this client during the session. For example, a shopping cart session bean or a student registration session bean is stateful because the session must keep track of which items or courses have been selected so far. A session bean class has a `Collection` type data member during a session but it does not have any permanent data storage to support.

We can see that a session bean can be accessed by any client just like a procedure or function. It does not keep any persistent state for a particular client during the conversation. It just takes the input from the client and sends the results back to the client. Another example of session bean can be an on-line session bean calculator or a logon verification bean.

4.2.3 EJB Web Service Components

Web services are getting popular recently thanks to the ubiquitous and platform-independent features of Web applications. It is important to make an existing EJB application a deployable Web service. EJB2.x can wrap a stateless session bean with a Web service endpoint interface, which can be accessed by any Web service client. EJB can also provide Web service via a Servlet front end by JAX-RPC. A stateless session bean also plays a front-end role in most of EJB distributed applications. The details of Web service implementation will be discussed in a separate chapter.

Here is a simple `hiUser` Web service endpoint interface implemented by a stateless session bean that extends `Remote` class in `rmi` package. The `hiUser` Web service endpoint wraps the session bean `HiUserBean`.

```
//HiUser.java is a web service endPoint interface for the
//stateless session bean
package hiUser;
import java.rmi.*;

public interface HiUser extends Remote{
    Public String hiUser(String user) throws RemoteException;
}

//HiUserBean.java is a stateless session bean implementation class
package hiUser;
import java.rmi.RemoteException;
import java.ejb.*;

public class HiUserBean implements SessionBean{
    public String hiUser(String user){
        Return "Hi " + user + "!";
    }
}
```

Here `HiUserBean` may be an existing stateless session bean. In order to provide a Web service, we must provide its WSDL file by a command line `wscompile`.

The steps to create a Web service endpoint are shown as follows:

1. Create all EJBs.
2. Create a `wsdl` Web service interface specification file by `wscompile` command.

3. Package the wsdl.xml file with all class files.
4. Assemble them into an ejb .ear file.
5. Deploy .ear file on a J2EE-compliant server.

A Web service can be accessed by any Web service client via HTTP protocol at a designated port. This session bean can also be accessed by a remote client if any additional remote and home interfaces are available.

4.2.4 Entity Bean

An entity bean represents some persistent data backed up by a database. Students, teachers, and courses are examples of entity beans. Each entity bean has an underlying table in the database and each instance of the bean is stored in a row of the table. An entity bean does not correspond to any specific client. It provides a shared access. It is supported by the EJB transaction service via its container. It has a persistent state with a unique primary key identifier that can be used by a client to locate a particular entity bean.

There are two types of entity beans: Bean Managed Persistence (BMP) entity beans and Container Managed Persistence (CMP) entity beans.

Here is an example of a student BMP entity bean. A database table must be created by the SQL:

```

CREATE TABLE student
  (id          VARCHAR(3) CONSTRAINT pk_student PRIMARY KEY,
   gpa        Number(2,1)
  );

//The following is the home interface for this student BMP entity
//bean.

import javax.ejb.*;
import java.util.*;
public interface StudentHome extends EJBHome {
    public Student create(String id, double gpa)
        throws RemoteException, CreateException;
    public Account findByPrimaryKey(String id)
        throws FinderException, RemoteException;
}

//The following code is the remote interface for this student BMP
//entity bean.

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Student extends EJBObject {
    public double getGpa() throws RemoteException;
}

//The following is the BMP implementation entity bean where SQL
//statements are explicitly included.

```

```

import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class StudentEJB implements EntityBean {
    private String id;
    private double gpa;
    private EntityContext context;
    private Connection con;
    private String dbName = "java:comp/env/jdbc/StudentDB";
    public double getGpa() {
        return gpa;
    }

    //The following ejb callback methods are executed by EJB
    //container. The Detailed references are in this chapter
    //reference.[1]
    //When a new bean instance is created the method ejbCreate()
    // is automatically called by the container to insert a row in a
    //corresponding table in the database.

    public String ejbCreate(String id, double gpa)
        throws CreateException {
        try {
            insertRow(id, gpa);
        } catch (Exception ex) {
            throw new EJBException("ejbCreate: ");
        }

        this.id = id;
        this.gpa = gpa;
        return id;
    }

    public String ejbFindByPrimaryKey(String primaryKey)
        throws FinderException {
        boolean result;
        try {
            result = selectByPrimaryKey(primaryKey);
        } catch (Exception ex) {
            throw new EJBException("ejbFindByPrimaryKey: ");
        }
        if (result) {
            return primaryKey;
        }
        else {
            throw new ObjectNotFoundException
                ("Row for id " + primaryKey + " not found.");
        }
    }
}

```

```

public void ejbRemove() {
    try {
        deleteRow(id);
    } catch (Exception ex) {
        throw new EJBException("ejbRemove: ");
    }
}

public void setEntityContext(EntityContext context) {
    this.context = context;
    try {
        makeConnection();
    } catch (Exception ex) {
        throw new EJBException("Failed to connect to
database.");
    }
}

public void ejbActivate() {
    id = (String)context.getPrimaryKey();
}

public void ejbPassivate() {
    id = null;
}

public void ejbLoad() {
    try {
        loadRow();
    } catch (Exception ex) {
        throw new EJBException("ejbLoad: ");
    }
}

public void ejbStore() {
    try {
        storeRow();
    } catch (Exception ex) {
        throw new EJBException("ejbLoad: " );
    }
}

public void ejbPostCreate(String id, double gpa) { }

void makeConnection() throws NamingException, SQLException
{ InitialContext ic = new InitialContext();
  DataSource ds = (DataSource) ic.lookup(dbName);
  con = ds.getConnection();
}

//The following methods are callback methods to invoke SQL
//statements to access database

```

```
void insertRow (String id, double gpa) throws SQLException {  
  
    String insertStatement = "insert into student values  
        (?,?)";  
    PreparedStatement prepStmt =  
        con.prepareStatement(insertStatement);  
    prepStmt.setString(1, id);  
    prepStmt.setDouble(2, gpa);  
    prepStmt.executeUpdate();  
    prepStmt.close();  
}  
  
void deleteRow(String id) throws SQLException {  
    String deleteStatement = "delete from student where id = ?  
    ";  
    PreparedStatement  
        prepStmt=con.prepareStatement(deleteStatement);  
    prepStmt.setString(1, id);  
    prepStmt.executeUpdate();  
    prepStmt.close();  
}  
  
boolean selectByPrimaryKey(String primaryKey) throws  
SQLException  
{  
    String selectStatement="select id "+"from student where id =  
    ? ";  
    PreparedStatement prepStmt =  
        con.prepareStatement(selectStatement);  
    prepStmt.setString(1, primaryKey);  
    ResultSet rs = prepStmt.executeQuery();  
    boolean result = rs.next();  
    prepStmt.close();  
    return result;  
}  
  
void loadRow() throws SQLException {  
    String selectStatement =  
        "select gpa " + "from student where id = ? ";  
    PreparedStatement prepStmt =  
        con.prepareStatement(selectStatement);  
    prepStmt.setString(1, this.id);  
    ResultSet rs = prepStmt.executeQuery();  
    if (rs.next()) {  
        this.gpa = rs.getDouble(2);  
        prepStmt.close();  
    }  
    else {  
        prepStmt.close();  
        throw new NoSuchEntityException(id + " not found.");  
    }  
}
```

```
void storeRow() throws SQLException {
    String updateStatement =
        "update student set gpa = ? " + "where id = ?";
    PreparedStatement prepStmt =
        con.prepareStatement(updateStatement);
    prepStmt.setDouble(1, gpa);
    prepStmt.setString(2, id);
    int rowCount = prepStmt.executeUpdate();
    prepStmt.close();
    if (rowCount == 0) {
        throw new EJBException("Store id " + id + " failed.");
    }
}
```

A developer for CMP entity beans does not need to write any database access code. However, he or she has to map the CMP entity bean to a table in the database along with related database operations at deployment time. On the other hand, BMP entity beans let developers have full control of database access for the bean. Of course, CMP is more portable and flexible than BMP and all details of database operations are specified in its DD.

4.2.5 Message-Driven Beans (MDB)

The MDB component is another type of Java server component running on the EJB container.

The MDB component works as a listener listening to a queue supported by Java Message Server (JMS).

Once a client sends a message to the queue, the MDB component will receive the message. It works in an asynchronous communication mode.

The MDB technology is available in J2EE 1.4 (EJB2.x). The MDB technology can work in two different ways: PTP (point to point) and publisher/subscriber. PTP works in one-to-one mode and publisher/subscriber works in a broadcasting (one to many) mode. The MDB technology works in an asynchronous fashion in that a notification can be received by an MDB component and its reactions could be immediate as long as MDB is active.

An MDB component works in the following way:

1. The container registers this MDB component with JMS.
 2. The JMS registers all JMS destinations (Topic for broadcasting or Queue for PTP) with Java Naming and Directory Interface (JNDI).
 3. The EJB container instantiates this MDB component.
 4. The client looks up the destination with the MDB.
 5. The client sends a message to the destination.
 6. The EJB container selects the corresponding MDB to consume the message.

MDB components work in a producer/consumer asynchronous mode and the message type can be text messages, object messages, stream messages, or byte messages. Messages are pushed and processed in a `MessageListener`'s method called `onMessage()`.

4.3 CONNECTION MODEL OF EJB

In this section, we will discuss the connection mechanisms among EJB components. We will explore how to use glue to tie up all basic component building blocks to generate new components or new applications. The connection model of EJB gives a guideline in design of individual EJB components, connections, and communications between related components as well [Component 2004].

In order for one client EJB component object to talk to another EJB component object, the client must get an object reference (pointer) to the target component and instantiate a remote instance that the

Remote Method Invocation (RMI) takes place afterward. This can be seen from the temperature converter example shortly.

There may be either synchronous or asynchronous connections between two EJB components via remote or local interfaces of the EJB components. An EJB component can also be accessed by a J2EE Web component such as a JavaBean component, a Servlet component, or a JSP component. An EJB component may also access other data objects of EIS by JDBC.

Figure 4.4 shows one front session bean connecting to two entity beans backed up by a database.

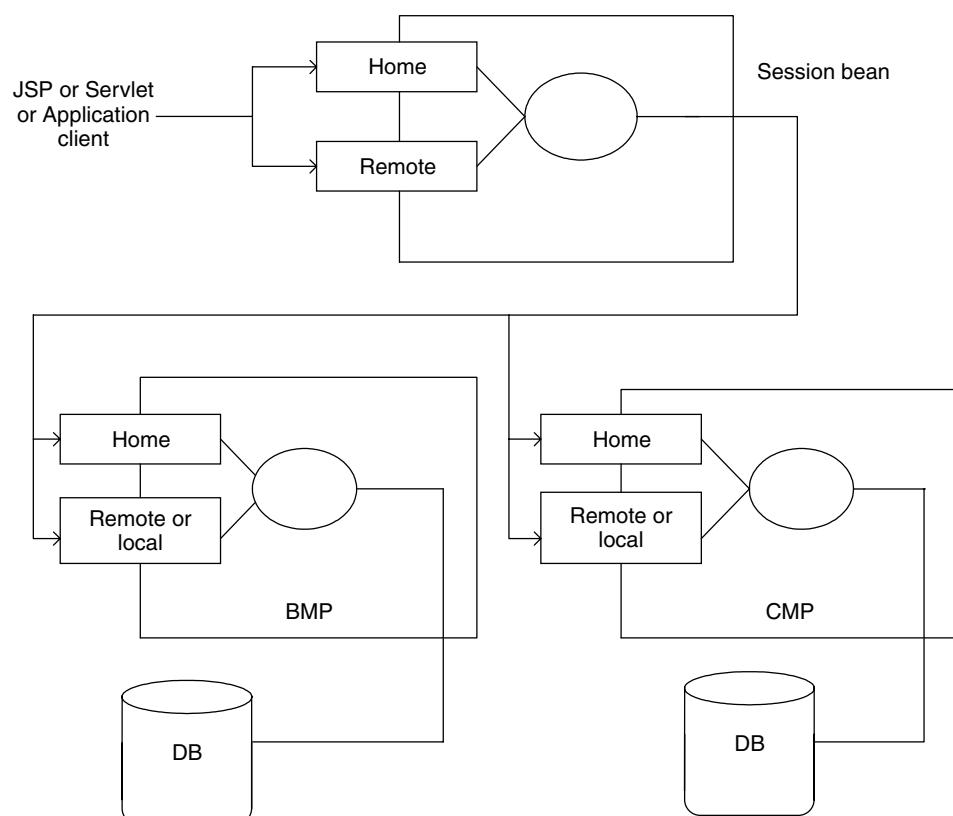


FIGURE 4.4. Connection between session beans and entity beans.

4.3.1 Tight Coupling Synchronous Connections

A synchronous invocation follows the request–response interaction mode. A client invokes a remote method in the interface of a target EJB component object. The component interface hides all details of implementations of business logic methods from the clients. After the target EJB component object completes its work, it may respond to the requests by returning the result or it may request other services from other components and wait for the result back and may forward it to the client afterward.

A typical example for this type of communication may be a shopping cart stateful session bean that sends a request to an order entity bean and wait for the order bean to process the order and to confirm the order.

Synchronous interaction leads to a tight coupling between EJB components. When a client component object initiates the request, the invocation thread is blocked from further processing until it receives a reply from the target EJB component. Most of the examples we have seen so far work in the synchronous mode.

4.3.2 Loosely Coupled Asynchronous Communication

An asynchronous communication involves a message-based communication between an EJB component and its client. A client makes a request to an EJB component but the client does not block itself. Instead, it continues its own process. There are two forms of asynchronous communications: queue-based PTP and publish-/subscribe-based message broadcasting. The queue-based asynchronous communication requires a message queue supported by JMS, which is between a client (message producer or sender) and a message consumer or message receiver. In publisher/subscriber mode, the consumer subscribes the topic into which the producer pushes the message and more than one subscriber (consumers) can receive the message from the topic.

In the asynchronous communication, clients and servers are loosely coupled, which results in a better throughput.

4.3.3 Remote Versus Local Communications

A client and its server component can be on the same machine or on different JVMs. If two EJB components are running on the same machine, there will be much less overhead by using local interfaces comparing to using remote interfaces. That is why EJB 2.x added the `localHome` and `localObject` interfaces in addition to remote interface. Some entity beans can provide both remote and local interfaces in EJB 2.x specification. A session bean may play a role of facade to entity beans, and an entity bean can be exposed directly to its clients or via a session bean or be connected to another entity bean. A local interface passes an argument to a method in *passing by reference* mode, while a remote interface does it in *passing by value* mode.

4.3.4 Object References to Entity Beans

In order to talk to a target entity bean, a stateless session bean must look up the deployed component according to its name registered in JNDI to get an object reference for it and may pass that reference to other beans. A stateful session bean may hold the reference to an entity bean in a conversational state. A session bean or MDB component can

hold the reference to another EJB component in an instance variable of its bean. A BMP entity bean can hold a reference to another entity bean in its nonpersistent field.

4.3.5 Association Connection Relationships between Entity Beans

Each entity bean has its persistent storage supported by a relational table. Each instance of an entity bean is stored as a row in the table regardless of BMP or CMP. The relationships between entity beans can be one to one, one to many, or many to many, just like the data relationships in a database reflected by E-R diagrams.

For example, the relationship between the student entity bean and the course entity bean is a many-to-many relationship. One student may take many courses and one course may be taken by many students. Let us implement these entity beans by CMP in EJB 2.x architecture. Assume these two CMP entity beans are deployed in one container so that local interface is used.

```
package student;
import java.util.*;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.naming.Context;
import javax.naming.InitialContext;

public abstract class StudentBean implements EntityBean {

    private EntityContext context;

    //StudentID and StudentName are CMP fields

    public abstract String getStudentID();          //primary key
    public abstract void    setStudentID(String id);

    public abstract String getName();
    public abstract void    setName(String lastName);

    //CMR(container-managed relationship) fields to course bean

    public abstract Collection getCourses();
    public abstract void    setCourses (Collection courses);

    //StudentBean next defines its business logic such that
    //getCourseList()
    //returns all corresponding courses this student has taken and
    //addCourse() will add a new course for this student.
```

```

public ArrayList getCourseList() {
    ArrayList list = new ArrayList();
    Iterator c = getCourses().iterator();
    while (c.hasNext()) {
        list.add(
            (LocalCourse)c.next());
    }
    return list;
}
public void addCourse (LocalCourse course) {
    getCourses().add(course);
}

//Student Local Home Interface
Here's the LocalStudentHome home interface for the StudentBean:
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;
public interface LocalStudentHome extends EJBLocalHome {
    public LocalStudent create (String studentID, String Name)
        throws CreateException;

    public LocalStudent findByPrimaryKey (
        String studentID)throws FinderException;
}

//The Student bean also defines a local interface. The bean's
//LocalStudent interface extends the EJBLocalObject interface
//not the EJBObject interface.

import java.util.ArrayList;
import javax.ejb.EJBLocalObject;

public interface LocalStudent extends EJBLocalObject {
    public String getStudentID();
    public String getName();
    public ArrayList getCourseList();
    public void addCourse(LocalCourse course);
    public void addCourse(String CourseKey); }
```

Here we only list the student CMP entity bean, and the course CMP entity bean has a very similar structure except the cmp and cmr fields. The Course bean includes equivalent methods to retrieve and set all students associated to a particular course instance.

```

public abstract Collection getStudents();
public abstract void setStudents(Collection students);
```

The DD specifies the enterprise bean relationship.

The deploy tool can be used to specify both the CMP and CMR fields, the bean-to-bean relationships, and the EJB SQL code for the finder methods.

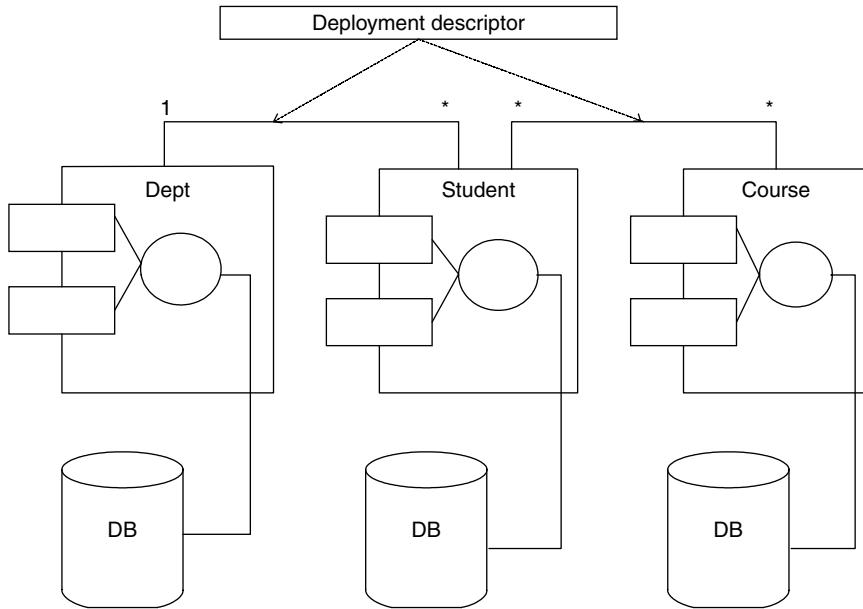


FIGURE 4.5. Association relationship between entity beans.

In Figure 4.5, there are three CMP entity beans. The **Dept** has one-to-many relationship to the **Student** entity bean and the **Student** entity bean has a many-to-many relationship to the **Course** entity bean. All the bean-to-bean relationship, **cmp** and **cmr** definitions, and implementation of some methods in SQL are defined by a deployer using deployment tool.

In comparison, BMP requires a lot of work to be done by a deployer, while CMP does not require the programmer provide any SQL coding.

4.4 DEPLOYMENT MODEL OF EJB

An EJB component is packaged as a **.jar** file, which is assembled in turn with other Web component packages (**.war**) and J2EE application client packages (**.jar**) in a J2EE application file (**.ear**).

The hierarchical structure of a J2EE application package is shown in Figure 4.6.

After creating the code for enterprise bean and its client, we need to compile them into class files. Next, we need to pack EJB components, Web components, or client into Java archive files (**.jar**) or Web archive file (**.war**) with their DD XML files and then assemble all of these archive files into an enterprise archive file (**.ear**) to be deployed on a server. The detail steps are shown in the lab practice in the next section.

A DD is a deployment definition file in XML format. It tells the types of the EJB, class names for remote interfaces, home interfaces, implementation beans, transaction management specification, access control security, and persistence property of entity beans. DD is created automatically by *deploytool* after the deployment wizard is completed.

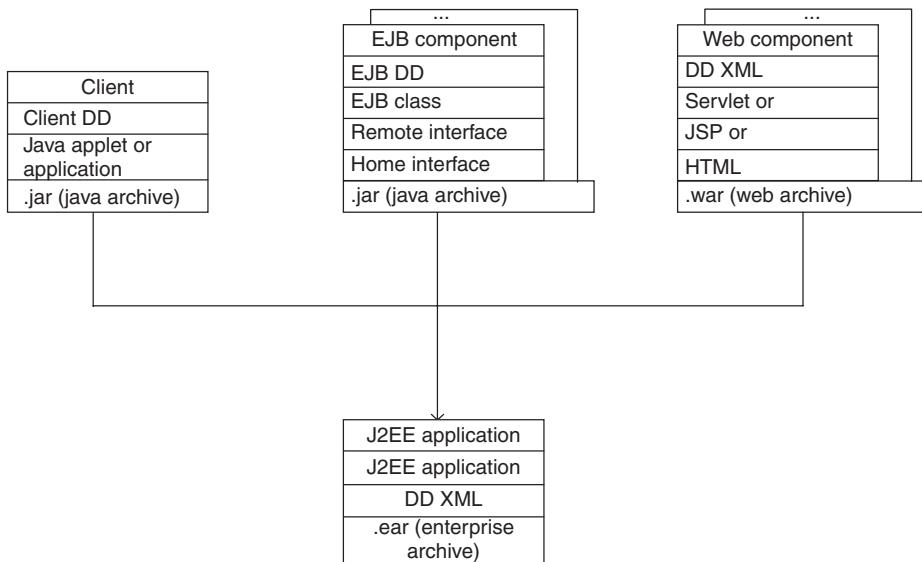


FIGURE 4.6. J2EE assembly and deployment.

The following is a partial content of a DD:

```

<?xml version="1.0">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>studentBean</ejb-name>
      <home>studenthome</home>
      <remote>student</remote>
      <ejb-class>Student</ejb-class>
      <persistence-type>Container</persistence-type>
      <pri-key-class>Integer</pri-key-class>
      ...
      <cmp-field><field-name>id</field-name></cmp-field>
      ...
      <cmp-field><field-name>name</field-name></cmp-field>
    </enterprise-beans>
    <assembly-descriptor>
      <security-role>
        ...
      </security-role>
      ...
    </assembly-descriptor>
  </ejb-jar>
  
```

4.5 EXAMPLES AND LAB PRACTICE

This section is designed to enhance the understanding of the EJB concepts by providing some concrete examples and step-by-step guidelines demonstrating how to build EJB

components, how to assemble and deploy EJB components, and how to build and run different client applications for these components. Here are two examples for lab practices.

Lab1 describes the steps to build a session bean called *temperature converter* component and to deploy it by the utility tool *deploytool* in J2EE 1.4.x. The client of this **temperature converter** EJB application is either a JSP web component that is accessed by a browser or a Java application client. Lab1 also demonstrates the running result of a web client and an application client of this EJB component.

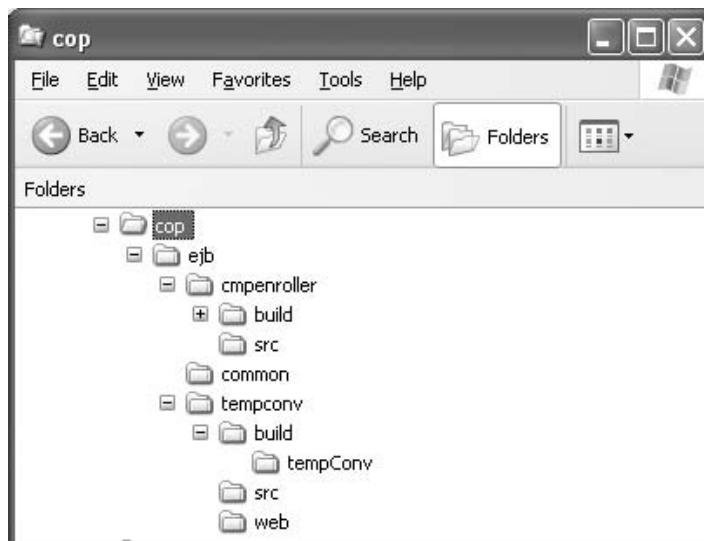
Lab2 is an implementation of a student registration system with one enrollment session EJB component and two CMP entity EJB components for student and course respectively. Lab2 shows the connection between the session bean and the entity beans, and association relationship between two entity beans.

4.5.1 Lab1: Temperature Converter Session Bean Component Development

Step 1: Installation and configuration for Lab1 and Lab2

1. Download J2EE 1.4 All-In-One bundle from www.java.sun.com.
2. Install J2EE 1.4 on your machine and ensure that the *<install_dir>/bin* directory is included in the environment's path.
3. All the source code and configuration files are available on this book's Wiley Website. Download the cop folder for Lab1 and Lab2 and put it under c:\.
4. Set up environment for Java-based *Ant* Build facility.

J2EE 1.4 All-In-One bundle includes Ant Build Tool, which is used in this lab for compiling Java source code and other processing. Ant has a relatively simple XML syntax. It is easy to learn and to use. In order to run the *Asant* scripts, the directory structure is built as below:



It is not strictly necessary to follow the suggested directory structure but it is strongly recommended until you are comfortable with it to make your own

modifications. Under the common folder there are two Ant scripts: `build.properties` and `targets.xml`.

Verify the variable in the `build.properties` based on the installation. Modify it if necessary.

By default, Ant looks for a file called `build.xml` that is in the working directory, for example `tempConv`. All the Java source code is in the `src` directory. JSP or HTML files are in the Web directory. When using Ant Build Tool to compile Java source code, a new directory `build` will be automatically created with all the class files.

Step 2: Coding the Enterprise Bean and client

The enterprise bean in this example includes the following code:

- Remote interface
- Home interface
- Enterprise bean class

```
//Remote interface: TempConv.java

package tempConv;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface TempConv extends EJBObject {
    public double fToC (double f ) throws RemoteException;
    public double cToF (double c) throws RemoteException;
}

//Home interface: TempConvHome.java

package tempConv;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface TempConvHome extends EJBHome {
    TempConv create() throws RemoteException, CreateException;
}

//Enterprise bean class: TempConvEJB.java

package tempConv;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.math.*;

public class TempConvEJB implements SessionBean {
```

```

public double cToF(double c) {
    return ((c*9.0/5.0 + 32)*100)/100.0;
}

public double fToC(double f) {
    return ((f-32)*5.0/9.0*100)/100.0;
}

public TempConvEJB() {}
public void ejbCreate() {}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext sc) {}
}

//Application Client for EJB: TempConvClient.java

import tempConv.TempConvHome;
import tempConv.TempConv;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class TempConvClient {
    public static void main(String[] args) {
        String output = "";
        String sDegree = "";
        double degree;
        DecimalFormat twoDigits = new DecimalFormat ("0.00");
        try {
            Context initial = new InitialContext();
            Context myEnv =
                (Context)initial.lookup("java:comp/env");
            Object objref = myEnv.lookup("ejb/ClientConv");
            TempConvHome home =
                (TempConvHome)PortableRemoteObject.narrow(objref,
                                                TempConvHome.class);
            TempConv converter = home.create();
            sDegree = JOptionPane.showInputDialog("Enter centigrade
                                                 degree");
            degree = Double.parseDouble( sDegree);
            output = sDegree + " centigrade degrees are " +
                     twoDigits.format (converter.cToF(degree)) +
                     " fahrenheit degrees" ;
            JOptionPane.showMessageDialog ( null, output,
                                         "results", JOptionPane.INFORMATION_MESSAGE);
            sDegree = JOptionPane.showInputDialog ( "Enter
                                                 fahrenheit degree" );
            degree = Double.parseDouble( sDegree);
        }
    }
}

```

```

        output = sDegree + " fahrenheit degrees are " +
                  twoDigits.format(converter.fToC(degree)) +
                  " centigrade degrees" ;
        JOptionPane.showMessageDialog ( null, output,
                  "results", JOptionPane.INFORMATION_MESSAGE);
        System.exit(0);
    }
    catch (Exception ex) {
        System.err.println("Caught an unexpected
                           exception!");
        ex.printStackTrace();
    }
}
}

<%-- Web Client for the EJB: Index.jsp --%>

<%@ page import="java.math.*"%>
<%@ page import="javax.naming.*"%>
<%@ page import="javax.ejb.*" %>
<%@ page import="javax.rmi.PortableRemoteObject" %>
<%@ page import="java.text.DecimalFormat" %>
<%@ page import="java.rmi.RemoteException" %>
<%@ page import="tempConv.TempConv" %>
<%@ page import="tempConv.TempConvHome" %>

<%
    private TempConv conv = null;

    public void jspInit() {
        try {
            InitialContext ic = new InitialContext();
            Object objRef = ic.lookup("java:comp/env/ejb/WebConv");
            TempConvHome home =
(TempConvHome)PortableRemoteObject.narrow(objRef,
TempConvHome.class);
            conv = home.create();
        } catch (RemoteException ex) {
            System.out.println("Couldn't create TempConv bean."+ex.getMessage());
        } catch (CreateException ex) {
            System.out.println("Couldn't create TempConv bean."+ex.getMessage());
        } catch (NamingException ex) {
            System.out.println("Unable to lookup home: "+
"WebConv "+
ex.getMessage());
        }
    }

    public void jspDestroy() {
        conv = null;
    }
}

```

```

%>
<html>
<head>
    <title>Temperature Converter</title>
</head>

<body bgcolor="white">
<h1><b><center> Temperature Converter </center></b></h1>
<hr>
<p>Enter a degree to convert:</p>
<form method="get">
<input type="text" name="degree" size="25">
<br>
<p>
<input type="submit" name="fToC" value="Fahrenheit to
    Centigrade">
<input type="submit" name="cToF" value="Centigrade to
    Fahrenheit">
</form>

<%
    DecimalFormat twoDigits = new DecimalFormat ("0.00");
    String degree = request.getParameter("degree");
    if ( degree != null && degree.length() > 0 ) {
        double d = Double.parseDouble(degree);
%>
<%
    if (request.getParameter("fToC") != null ) {
%>
        <p>
            <%= degree %> fahrenheit degrees are
            <%= twoDigits.format(conv.fToC(d)) %>
            centigrade degrees.
<%
    }
%>
<%
    if (request.getParameter("cToF") != null ) {
%>
        <p>
            <%= degree %> centigrade degrees are
            <%= twoDigits.format (conv.cToF(d)) %>
            fahrenheit degrees .
<%
    }
%>
<%
}
%>
</body>
</html>

```

Step 3: Compiling all source code

Compile the source files at command prompt in c:\cop\ejb\tempconv directory. Type

>asant build

```
C:\cop\ejb\tempconv>asant build
Buildfile: build.xml
init:
prepare:
[mkdir] Created dir: C:\cop\ejb\tempconv\build
build:
[javac] Compiling 4 source files to C:\cop\ejb\tempconv\build
BUILD SUCCESSFUL
Total time: 11 seconds
C:\cop\ejb\tempconv>cd ..
```

A new directory **build** is created with class files.

Step 4: Deployment

1. Start J2EE Application Server

On Windows Start menu, choose Program | Sun Microsystem | Application Server | Start Default Serve to start J2EE Application Server.

2. Start deploytool

On Windows Start menu, choose Program | Sun Microsystem | Application Server | deploytool to start deploytool utility.

3. Creating the J2EE Application

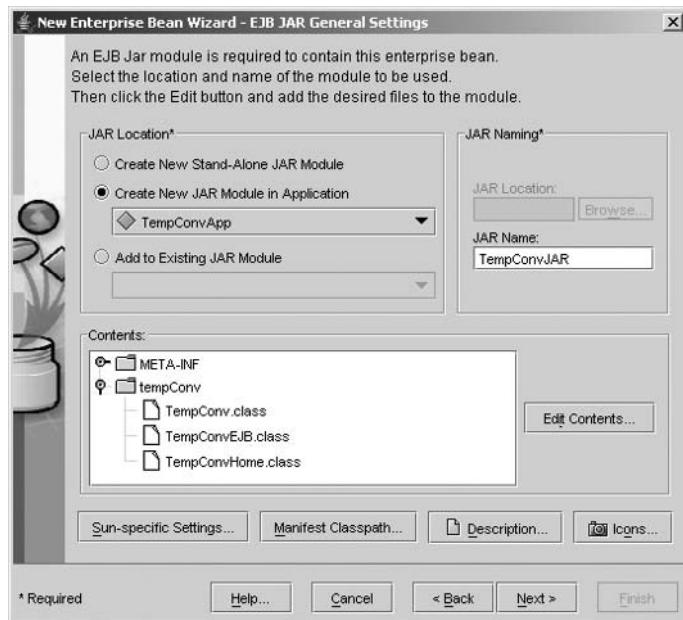
- In *deploytool*, select File | New | Application.
- Click Browse.
- In the file chooser, navigate to c:\cop\ejb\tempconv.
- In the File Name field, enter TempConvApp.ear.
- Click New Application and OK.

4. Packaging the EJB

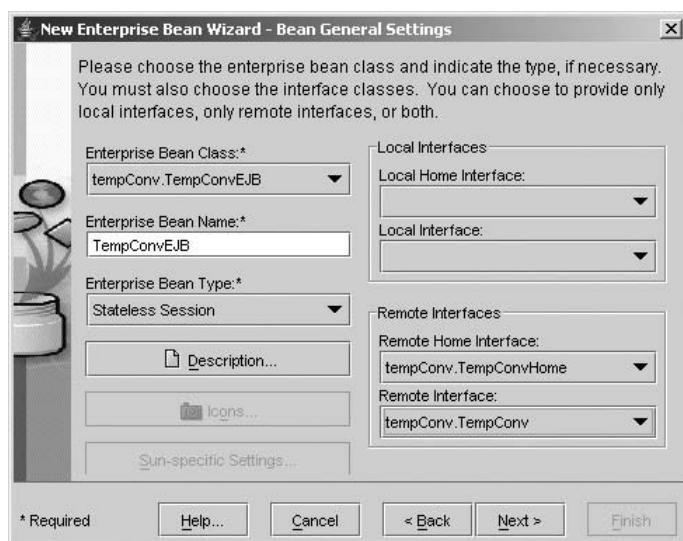
To package an enterprise bean, we run the **New Enterprise Bean** wizard of the *deploytool* utility, which will create the bean's deployment descriptor, package the deployment descriptor and the bean's classes in an EJB JAR file, and insert the EJB JAR file into the application's **TempConvApp.ear** file.

To start the **New Enterprise Bean** wizard, select File | New | Enterprise Bean.

- In the EJB JAR General Settings dialog box, select the Create New JAR Module in Application radio button.
- In the combo box, select TempConvApp.
- In the JAR Display Name field, enter TempConvJAR.
- Click Edit Contents.
- In the tree under Available Files, go to c:\cop\ejb\tempconv\build directory.
- Select TempConv folder from the Available Files tree and click Add.
- Click OK. And Next.



- In the Bean General Settings dialog box select `tempConv.TempConvEJB` in the Enterprise Bean Class combo box. Verify `TempConvEJB` in the Enterprise Bean Name field. Under Bean Type, select the Stateless Session radio buttons.
- Select `tempConv.TempConvHome` in the Remote Home Interface combo box, select `tempConv.TempConv` in the Remote Interface combo box, click Next.
- In the Expose as Web Service Endpoint dialog box, select the No radio button, click Next and Finish.



After the packaging process, you can view the deployment descriptor by selecting Tools | Descriptor Viewer.

```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar version="2.1"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"      >
<display-name>TempConvJAR</display-name>
<enterprise-beans>
  <session>
    <ejb-name>TempConvEJB</ejb-name>
    <home>tempConv.TempConvHome</home>
    <remote>tempConv.TempConv</remote>
    <ejb-class>tempConv.TempConvEJB</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Bean</transaction-type>
    <security-identity> <use-caller-identity>
      </use-caller-identity></security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

5. Packaging the Application Client

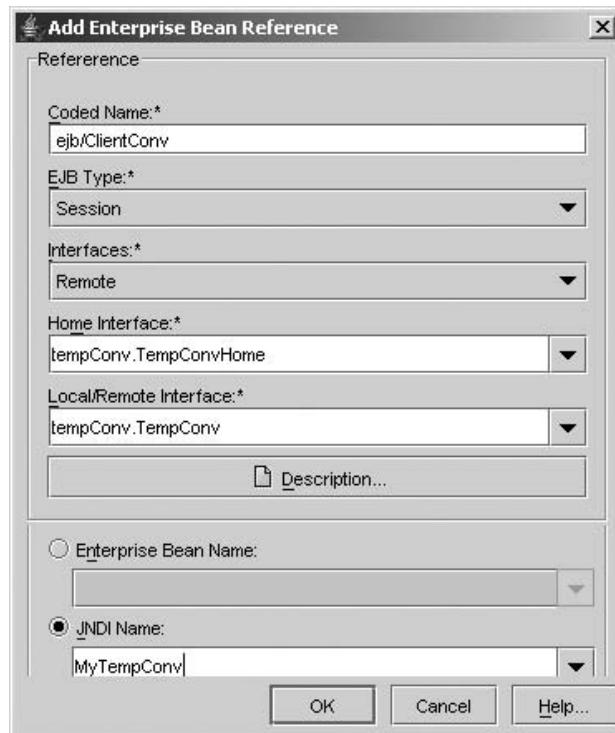
To start the New Application Client wizard, select File | New | Application Client.

- In Jar File Contents dialog box, select Create New AppClient Module in Application radio button.
- Select TempConvApp in the combo box.
- Enter TempConvClient in the AppClient Name field.
- Click Edit Contents button.
- In the tree under Available Files, go to c:\cop\ejb\tempconv\build directory.
- Select the TempConvClient.class file and click Add.
- Click OK and Next.
- Select TempConvClient in the Main Class combo box in the General dialog box.
- Select use container-managed authentication in the Callback Handler Class combo box.
- Now, click Next and Finish.



Specifying the Application Client's Enterprise Bean Reference:

- In the tree, select TempConvClient.
- Select the EJB Refs tab and click Add.
- Type ejb/ClientConv in the Coded Name column, select Session in the Type field, and select Remote in the Interfaces field.
- Type tempConv.TempConvHome in the Home Interface field and type tempConv.TempConv in the Local/Remote Interface field.
- Select JNDI Name radio button in the Target EJB box, enter MyTempConv and click OK.



After the packaging process, you can view the deployment descriptor by selecting Tools | Descriptor Viewer.

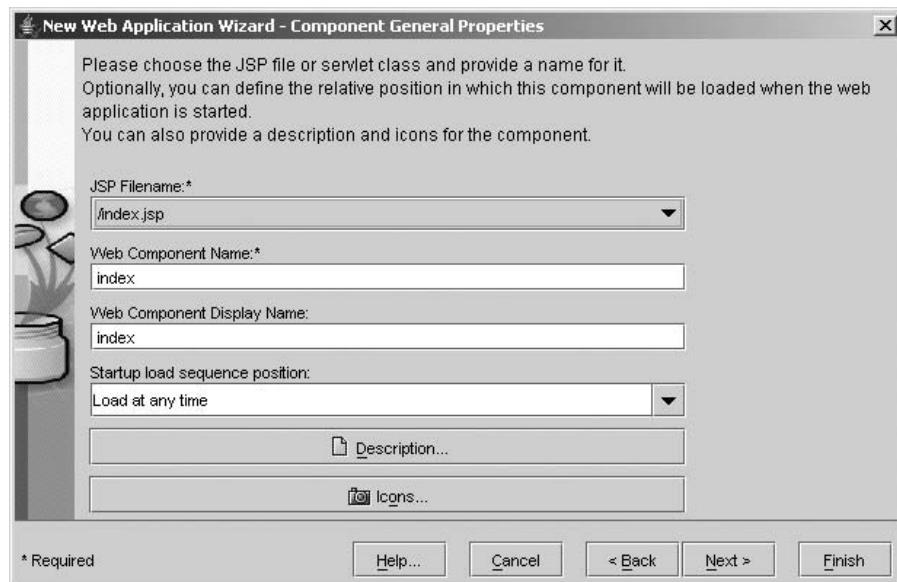
```
<?xml version='1.0' encoding='UTF-8'?>
<application-client      version="1.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee-
        http://java.sun.com/xml/ns/j2ee/application-
        client_1_4.xsd"
    >      <display-name>TempConvClient</display-name>
<ejb-ref>
    <ejb-ref-name>ejb/ClientConv</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>tempConv.TempConvHome</home>
    <remote>tempConv.TempConv</remote>
</ejb-ref>
</application-client>
```

6. Packaging the Web Client

To package a Web client, we can first run the New Web Component Wizard, which will create a web application deployment descriptor. Next, add all component files to a war file and add the war file to the application's TempConv.ear file.

Build the web component by the New Web Component wizard and select File | New | Web Component. See the following dialog boxes:

- In the WAR File dialog box, select Create New WAR Module in Application.
- Select TempConvApp in the combo box.
- Enter TempConvWAR In the WAR Name field.
- Click Edit Contents.
- In the tree under Available Files, go to c:\cop\ejb\tempconv\web directory.
- Select index.jsp and click Add.
- Click OK and Next.
- In the Choose Component Type dialog box, select JSP radio button. Click Next
- In the Component General Properties dialog box, select index.jsp in the JSP FileName combo box.
- Click Finish.



Specifying the Web Client's Enterprise Bean Reference:

- In the tree, select TempConvWAR.
- Select the EJB Refs tab.
- Click Add.
- In the Coded Name field, enter ejb/WebConv.
- Select Session in the Type field.
- Select Remote in the Interfaces field.
- Type tempConv.TempConvHome in the Home Interface field and type tempConv.TempConv in the Local/Remote Interface field.
- Select JNDI Name radio button for the Target EJB, enter MyTempConv, and click OK.

After the packaging process, you can view the following deployment descriptor by selecting Tools | Descriptor Viewer.

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"      >
<display-name>TempConvWAR</display-name>
<servlet>      <display-name>index</display-name>
                <servlet-name>index</servlet-name>
                <jsp-file>/index.jsp</jsp-file>
</servlet>
<ejb-ref>      <ejb-ref-name>ejb/WebConv</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
```

```

<home>tempConv.TempConvHome</home>
<remote>tempConv.TempConv</remote>
</ejb-ref>
</web-app>

```

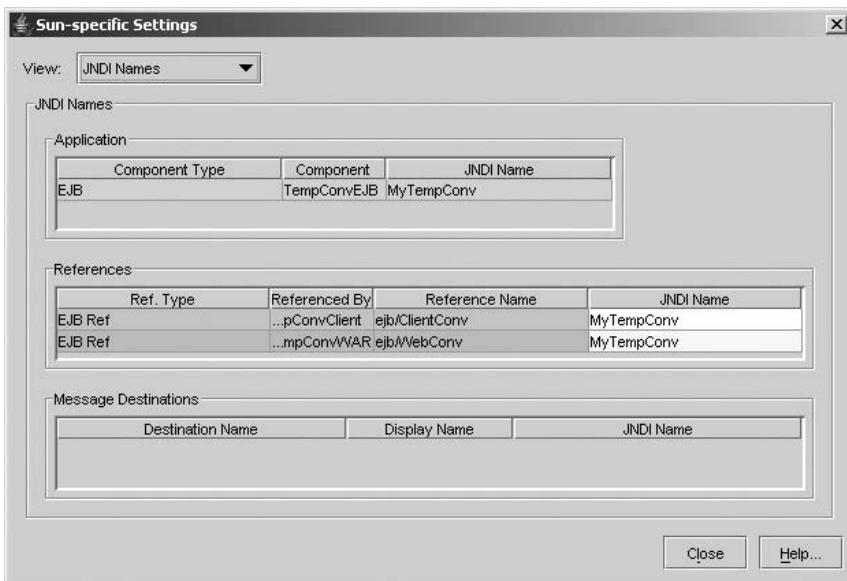
Specifying the JNDI Names:

Follow the following steps to map the enterprise bean references used by the clients to the JNDI name of the bean.

- In the tree, select TempConvApp.
- Click Sun-Specific Settings button.
- Choose JNDI Names for View.
- To specify a JNDI name for the bean, find the TempConvEJB component in the Application table and enter MyTempConv in the JNDI Name column.
- To map the references, in the References table enter MyTempConv in the JNDI Name for each row.

Specifying the Context Root:

- In the tree, select TempConvWAR.
- Click General tab and enter tempconv in the Context Root field, so tempconv becomes the context root in URL.
- Save the application file.

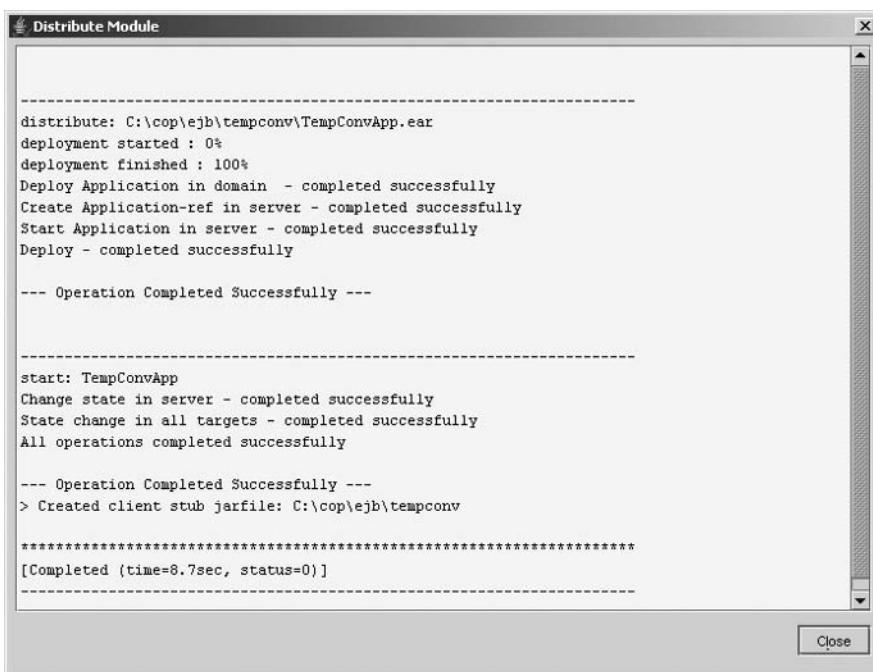
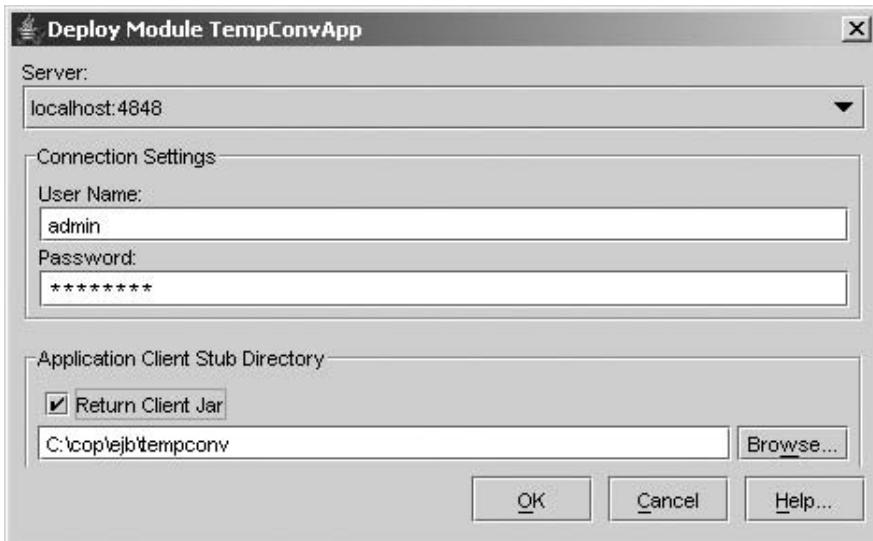


7. Deploying the J2EEApplication

At this time, the J2EE application contains all components and it is ready for deployment.

- Select the TempConvApp application.
- Select Tools | Deploy.
- Select the checkbox labeled Return Client Jar.

- Verify the full path name for the file TempConvAppClient.jar so that it will reside in the c:\cop\ejb\tempconv subdirectory. The TempConvAppClient.jar file has the EJB stub that enables remote access to TempConvEJB.
- Click OK.
Notes: To undeploy an application
- Under the Servers in the tree, Click localhost:4848.
- Select the Deployed Object and click undeploy.

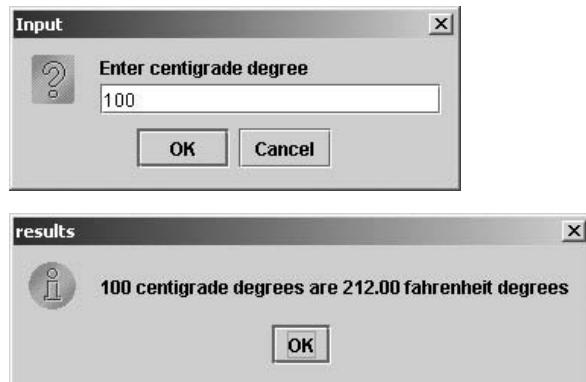


Step 5: Running the J2EE Application Client and Web Client

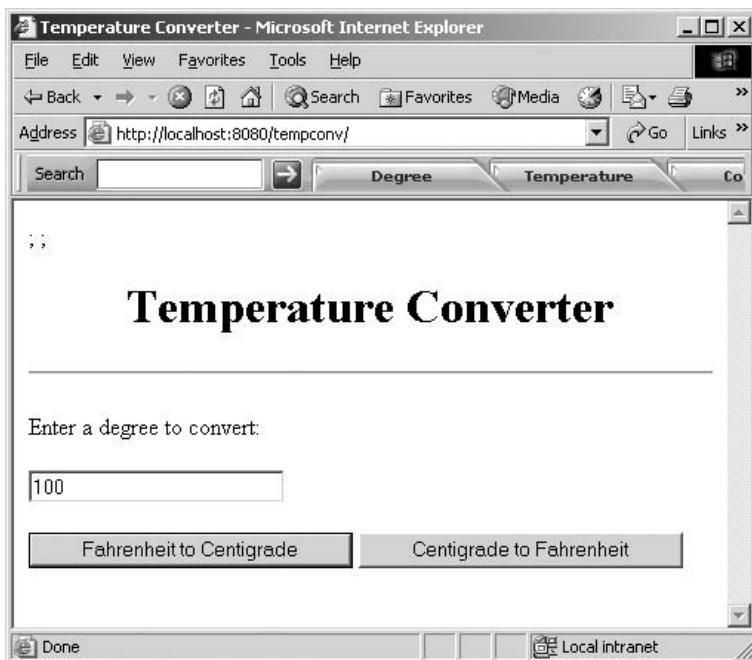
In a terminal window, go to the c:\cop\ejb\tempconv directory, check TempConvApp.ear and TempConvAppClient.jar files and then type

>appclient -client TempConvAppClient.jar

You will see the screen shown in the figure after entering 100 in the input field and clicking OK.



To run the Web client, point your browser at the following URL. Replace <host> with the name of the host running the J2EE server. If your browser is running on the same host as the J2EE server, you may replace <host> with localhost. By default your URL is <http://localhost:8080/tempconv>.



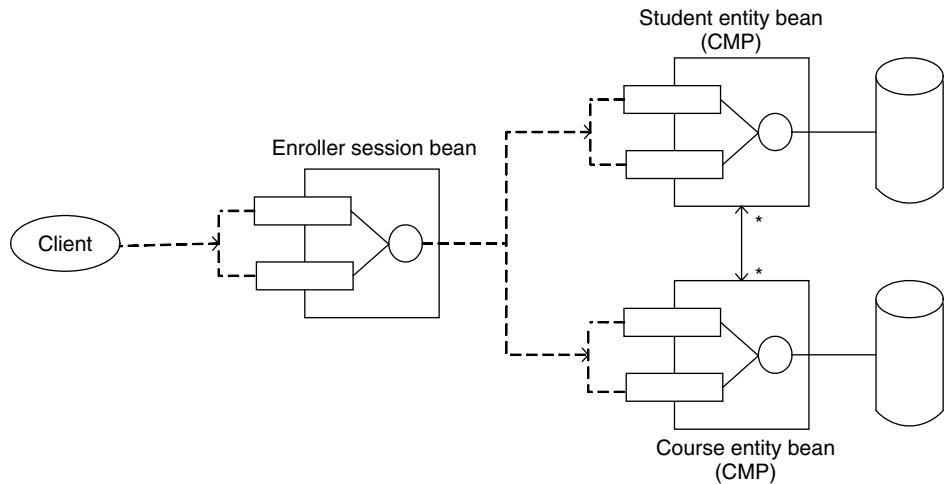
You should see the screen shown in the figure after entering 100 in the input field and clicking {Fahrenheit to Centigrade}.



4.5.2 Lab2: A Complete J2EE Application with One Session Bean and Two Entity Beans

The Lab2 implements a student registration application “EnrollerApp”, which consists of a front session bean and two backend CMP entity beans supported by database tables. It can add new course or new student into the registration system, remove a course, or remove a student. It can also register a student with a number of courses or drop a student from one or many courses registered. The system can also report some statistical data of registration such as the enrollment of a specific course.

This EnrollerApp application has four components. The EnrollerAppClient component is a J2EE application client that accesses the EnrollerEJB session bean through the bean’s remote interface. The EnrollerBean accesses two entity beans – StudentBean and CourseBean – through their local interfaces. The entity beans use container-managed persistence and relationships. The StudentBean and CourseBean entity beans have a many-to-many relationship. Each bean has a relationship field whose value identifies the related bean instance. The diagram below shows the components and their relationships in the EnrollerApp application. The dotted lines represent the access gained through invocations of the JNDI lookup method. The solid lines represent the container-managed relationships [Armstrong 2004; Component 2004; Pawlan 2000].



The example in this Lab is tested with a **PointBase** database, which provides persistent storage for application data. The **PointBase** is included in the J2EE SDK.

Step 1: Develop Beans and their client

Student entity bean component includes the following code:

- Local interface: `LocalStudent.java`
- Local home interface: `LocalStudentHome.java`
- Entity bean class: `StudentBean.java`

```
//LocalStudent.java is a Local interface for student CMP entity
//bean, It can be a remote interface if its client is remote.
```

```
package cmppack;

import java.util.*;
import javax.ejb.*;

public interface LocalStudent extends EJBLocalObject {

    public String getStudentId();
    public String getName();
    public Collection getCourses();
}

//LocalStudentHome.java is a Home interface of student CMP
//entity bean
//It can be defined as a remoteHome interface if accessed by a
//remote client

package cmppack;
```

```

import java.util.*;
import javax.ejb.*;

public interface LocalStudentHome extends EJBLocalHome {

    public LocalStudent create (String id, String name)
        throws CreateException;

    public LocalStudent findByPrimaryKey (String id)
        throws FinderException;

    public Collection findByName(String name)
        throws FinderException;

    public Collection findAll()
        throws FinderException;
}

//StudentBean.java is a CMP entity bean class implementation

package cmppack;

import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public abstract class StudentBean implements EntityBean {

    private EntityContext context;

    // Access methods for persistent fields, implicit definition of
    // class members

    public abstract String getStudentId();
    public abstract void setStudentId(String id);

    public abstract String getName();
    public abstract void setName(String name);

    // Access methods for relationship fields.

    public abstract Collection getCourses();
    public abstract void setCourses(Collection courses);

    // Select methods, Business methods, EntityBean methods

    public String ejbCreate (String id, String name )
        throws CreateException {
        setStudentId(id);
        setName(name);
        return null;
}

```

```

public void ejbPostCreate (String id, String name)
    throws CreateException { }

public void setEntityContext(EntityContext ctx) {
    context = ctx;
}

public void unsetEntityContext() {
    context = null;
}

public void ejbRemove() {}
public void ejbLoad() {}
public void ejbStore() {}
public void ejbPassivate() {}
public void ejbActivate() {}

} // StudentBean class

```

Course entity bean component includes the following code

- Entity bean class: CourseBean.java
- Local home interface: LocalCourseHome.java
- Local interface: LocalCourse.java

```

//LocalCourse.java

package cmppack;

import java.util.*;
import javax.ejb.*;

// Local interface of course CMP entity bean

public interface LocalCourse extends EJBLocalObject {

    public String getCourseId();
    public String getTitle();
    public int getMaxEnrollment();
    public int getCurrentEnrollment();
    public Collection getStudents();

    public void addStudent(LocalStudent Student);
    public void removeStudent(LocalStudent Student);

}

//LocalCourseHome.java

package cmppack;

import java.util.*;

```

```

import javax.ejb.*;

// Home interface of course CMP entity bean

public interface LocalCourseHome extends EJBLocalHome {

    public LocalCourse create (String id, String title, int
        MaxEnrollment, int CurrentEnrollment) throws
        CreateException;

    public LocalCourse findByPrimaryKey (String id)
        throws FinderException;
}

//CourseBean.java

package cmppack;

import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public abstract class CourseBean implements EntityBean {

    private EntityContext context;

    // Access methods for persistent fields: courseId, title,
    // maxEnrollment, currentEnrollment

    public abstract String getCourseId();
    public abstract void setCourseId(String id);

    public abstract String getTitle();
    public abstract void setTitle(String title);

    public abstract int getMaxEnrollment();
    public abstract void setMaxEnrollment(int maxenroll);

    public abstract int getCurrentEnrollment();
    public abstract void setCurrentEnrollment(int currenroll);

    // Access methods for relationship fields

    public abstract Collection getStudents();
    public abstract void setStudents(Collection students);

    // Business methods

    public void addStudent(LocalStudent student) {
        try {
            Collection students = getStudents();
            students.add(student);
            setCurrentEnrollment(getCurrentEnrollment() + 1);
        }
    }
}

```

```

        } catch (Exception ex) {
            throw new EJBException(ex.getMessage());
        }
    }

    public void removeStudent(LocalStudent student) {
        try {
            Collection students = getStudents();
            students.remove(student);
            setCurrentEnrollment(getCurrentEnrollment() - 1);
        } catch (Exception ex) {
            throw new EJBException(ex.getMessage());
        }
    }

    // EntityBean container callback methods

    public String ejbCreate (String id, String title, int
                           maxenroll, int currenroll)
        throws CreateException {
        setCourseId(id);
        setTitle(title);
        setMaxEnrollment(maxenroll);
        setCurrentEnrollment(currenroll);
        return null;
    }

    public void ejbPostCreate (String id, String title, int
                           maxenroll, int currenroll)
        throws CreateException { }

    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }

    public void unsetEntityContext() {
        context = null;
    }

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}

} // CourseBean class

```

Enroller session bean includes the following code:

- Remote interface: Enroller.java
- Remote home interface: EnrollerHome.java
- Session bean class: EnrollerBean.java

- StudentDetails.java
- CourseDetails.java

```
//Enroller.java is a Remote interface of Enroller session bean

package enroller;

import java.util.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import cmppack.*;

public interface Enroller extends EJBObject {

    // Students
    public void createStudent(StudentDetails details)
        throws RemoteException;

    public void addStudentToCourse(String studentId, String
        courseId)
        throws RemoteException;

    public void removeStudentFromCourse(String studentId,
        String courseId)
        throws RemoteException;

    public ArrayList getStudentsOfCourse(String courseId)
        throws RemoteException;

    // Courses
    public void createCourse(CourseDetails details)
        throws RemoteException;

    public void removeCourse(String courseId)
        throws RemoteException;

    public CourseDetails getCourse(String courseId)
        throws RemoteException;
}

//EnrollerHome.java is a Remote home interface of Enroller
//session bean

package enroller;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface EnrollerHome extends EJBHome {

    Enroller create() throws RemoteException, CreateException;
```

```
}

//EnrollerBean.java

package enroller;

import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import cmppack.*;

public class EnrollerBean implements SessionBean {

    private LocalStudentHome studentHome = null;
    private LocalCourseHome courseHome = null;

    // student business methods

    public void createStudent(StudentDetails details) {
        try {
            LocalStudent student =
                studentHome.create(details.getId(),
                    details.getName());
        } catch (Exception ex) {
            throw new EJBException(ex.getMessage());
        }
    }

    // add a student to a course

    public void addStudentToCourse(String studentId, String
        courseId) {
        try {
            LocalCourse course =
                courseHome.findByPrimaryKey(courseId);
            LocalStudent student =
                studentHome.findByPrimaryKey(studentId);
            course.addStudent(student);

        } catch (Exception ex) {
            throw new EJBException(ex.getMessage());
        }
    }

    // drop student from a course

    public void removeStudentFromCourse(String studentId,
        String courseId) {
        try {
            LocalStudent student =
                studentHome.findByPrimaryKey(studentId);
```

```

        LocalCourse course =
            courseHome.findByPrimaryKey(courseId);
        course.removeStudent(student);
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

// get student list for a course

public ArrayList getStudentsOfCourse(String courseId) {
    Collection students = null;

    try {
        LocalCourse course =
            courseHome.findByPrimaryKey(courseId);
        students = course.getStudents();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }

    return copyStudentsToDetails(students);
} // getStudentsOfCourse

// Course business methods

public void createCourse(CourseDetails details) {
    try {
        LocalCourse course =
            courseHome.create(details.getId(),
                details.getTitle(), details.getMaxEnrollment(),
                details.getCurrentEnrollment());
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

public void removeCourse(String courseId) {
    try {
        LocalCourse course =
            courseHome.findByPrimaryKey(courseId);
        course.remove();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

public CourseDetails getCourse(String courseId) {
    CourseDetails courseDetails = null;
    try {
        LocalCourse course =
            courseHome.findByPrimaryKey(courseId);
        courseDetails = new CourseDetails (courseId,

```

```
        course.getTitle(), course.getMaxEnrollment(),
        course.getCurrentEnrollment());
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
    return courseDetails;
}

// SessionBean methods

public void ejbCreate() throws CreateException {
    try {
        studentHome = lookupStudent();
        courseHome = lookupCourse();
    } catch (NamingException ex) {
        throw new CreateException(ex.getMessage());
    }
}

public void ejbActivate() {
    try {
        studentHome = lookupStudent();
        courseHome = lookupCourse();
    } catch (NamingException ex) {
        throw new EJBException(ex.getMessage());
    }
}

public void ejbPassivate() {

    studentHome = null;
    courseHome = null;
}

public EnrollerBean() {}
public void ejbRemove() {}
public void setSessionContext(SessionContext sc) {}

// Private methods

private LocalStudentHome lookupStudent() throws
    NamingException {

    Context initial = new InitialContext();
    Object objref =
    initial.lookup("java:comp/env/ejb/SimpleStudent");
    return (LocalStudentHome) objref;
}

private LocalCourseHome lookupCourse() throws
    NamingException {
    Context initial = new InitialContext();
    Object objref =
```

```

        initial.lookup("java:comp/env/ejb/SimpleCourse");
        return (LocalCourseHome) objref;
    }

private ArrayList copyStudentsToDetails(Collection students)
{
    ArrayList detailsList = new ArrayList();
    Iterator i = students.iterator();

    while (i.hasNext()) {
        LocalStudent student = (LocalStudent) i.next();
        StudentDetails details = new
            StudentDetails(student.getStudentId(),
                           student.getName());
        detailsList.add(details);
    }

    return detailsList;
} // copyStudentsToDetails
} // EnrollerBean

//StudentDetails.java

package enroller;

public class StudentDetails implements java.io.Serializable {

    private String id;
    private String name;

    public StudentDetails (String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        String s = "Student Id: " + id + "    Name: " + name +
                  "\n"; return s;
    }
}

} // StudentDetails

//CourseDetailes.java

```

```
package enroller;

public class CourseDetails implements java.io.Serializable {

    private String id;
    private String title;
    private int maxenroll;
    private int currenroll;

    public CourseDetails (String id, String title, int
        maxenroll, int currenroll) {
        this.id = id;
        this.title = title;
        this.maxenroll = maxenroll;
        this.currenroll = currenroll;
    }

    public String getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public int getMaxEnrollment() {
        return maxenroll;
    }

    public int getCurrentEnrollment() {
        return currenroll;
    }

    public String toString() {
        String s = "Course Id: " + id + "    Course Title: "
            + title + "\n" + "Max #: " + maxenroll + " Current
            #: " + currenroll + "\n";
        return s;
    }
} // CourseDetails

//Enroller Client: EnrollerClient.java

package client;

import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import cmppack.*;
import enroller.*;
import javax.swing.*;
```

```

public class EnrollerClient {

    public static void main(String[] args) {
        String output;
        try {
            Context initial = new InitialContext();
            Object objref =
                initial.lookup("java:comp/env/ejb/SimpleEnroller");
            EnrollerHome home =
                (EnrollerHome)PortableRemoteObject.narrow(objref,
                                                EnrollerHome.class);
            Enroller myEnroller = home.create();
            output = createCourses(myEnroller);
            output = output + register(myEnroller);
            output = output + getClassList(myEnroller);
            JOptionPane.showMessageDialog(null, output,
                                         "Results", JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);

        } catch (Exception ex) {
            System.err.println("Caught an exception:");
            ex.printStackTrace();
        }
    } // main

    private static String getClassList(Enroller myEnroller) {
        String s = "Class List \n";
        try {
            ArrayList studentList;
            CourseDetails courseInfo;
            courseInfo = myEnroller.getCourse("C1");
            s = s + courseInfo.toString();
            studentList = myEnroller.getStudentsOfCourse("C1");
            s = s + dispDetailsList(studentList) + "\n";
            courseInfo = myEnroller.getCourse("C2");
            s = s + courseInfo.toString();
            studentList = myEnroller.getStudentsOfCourse("C2");
            s = s + dispDetailsList(studentList) + "\n";

        } catch (Exception ex) {
            System.err.println("Caught an exception:");
            ex.printStackTrace();
        }
        return s;
    } // getClassList

    private static String dispDetailsList(ArrayList list) {
        String s = "";
        Iterator i = list.iterator();
        while (i.hasNext()) {
            Object details = (Object)i.next();
            s = s + details.toString();
        }
    }
}

```

```
        }
    return s;
} // printDetailsList

// Create Courses

private static String createCourses(Enroller myEnroller) {
    String s = "";
    try {
        myEnroller.createCourse(new CourseDetails(
            "C1", "JAVA Programming", 50, 0));
        s = "Create course C1 - JAVA Programming \n";
        myEnroller.createCourse(new CourseDetails(
            "C2", "Component Oriented Programming", 50, 0));
        s = s + "Create course C2 - Component Oriented
Programming \n\n";
    } catch (Exception ex) {
        System.err.println("Caught an exception:");
        ex.printStackTrace();
    }
    return s;
}

// Create students and add to courses

private static String register(Enroller myEnroller) {
    String s = "";
    try {
        myEnroller.createStudent(new StudentDetails(
            "S1", "Dan Jones"));
        myEnroller.addStudentToCourse("S1", "C1");
        myEnroller.addStudentToCourse("S1", "C2");
        s = s + "Create student S1 - Dan Jones \n" +
            "Add student to course C1, C2 \n";

        myEnroller.createStudent(new StudentDetails(
            "S2", "Vivian Smith"));
        myEnroller.addStudentToCourse("S2", "C1");
        myEnroller.addStudentToCourse("S2", "C2");
        s = s + "Create student S2 - Vivian Smith \n" +
            "Add student to course C1, C2 \n";

        myEnroller.createStudent(new StudentDetails(
            "S3", "Bob Roberts"));
        myEnroller.addStudentToCourse("S3", "C1");
        s = s + "Create student S3 - Bob Roberts \n" +
            "Add student to course C1 \n\n";
        myEnroller.removeStudentFromCourse("S2", "C1");
        s = s + "Drop student S2 from course C1.\n\n";
    } catch (Exception ex) {
        System.err.println("Caught an exception:");
        ex.printStackTrace();
    }
}
```

```

        }
        return s;
    }
}

```

Step 2: Set up environment and getting started

See Lab1 for environment setup and start J2EE Application Server.

To start PointBase on Windows, from start menu, choose Program | Sun Microsystems | Application Server | Start PointBase.

Step 3: Compiling the Source Files

To compile the source files, open a terminal window, go to c:\cop\ejb\cmpenroller directory and type the following command:

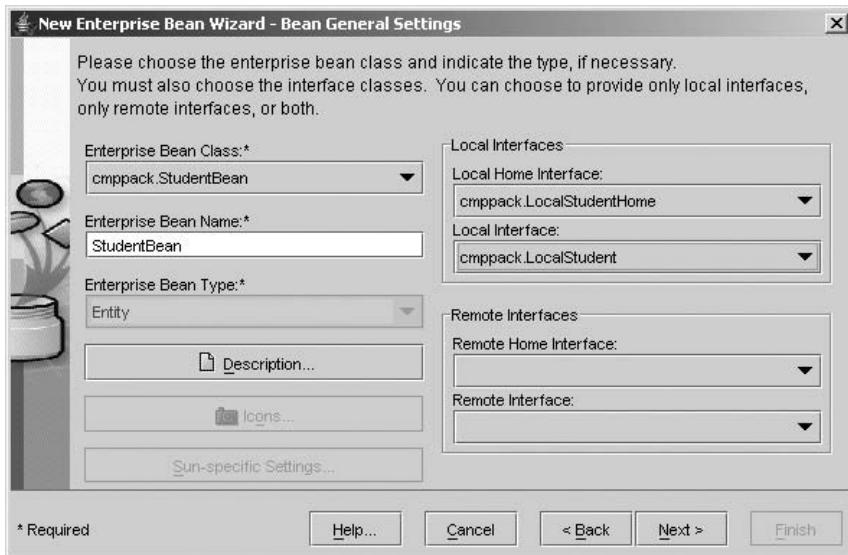
>asant build

A new directory “build” with all the class files will be created.

Step 4: Deployment

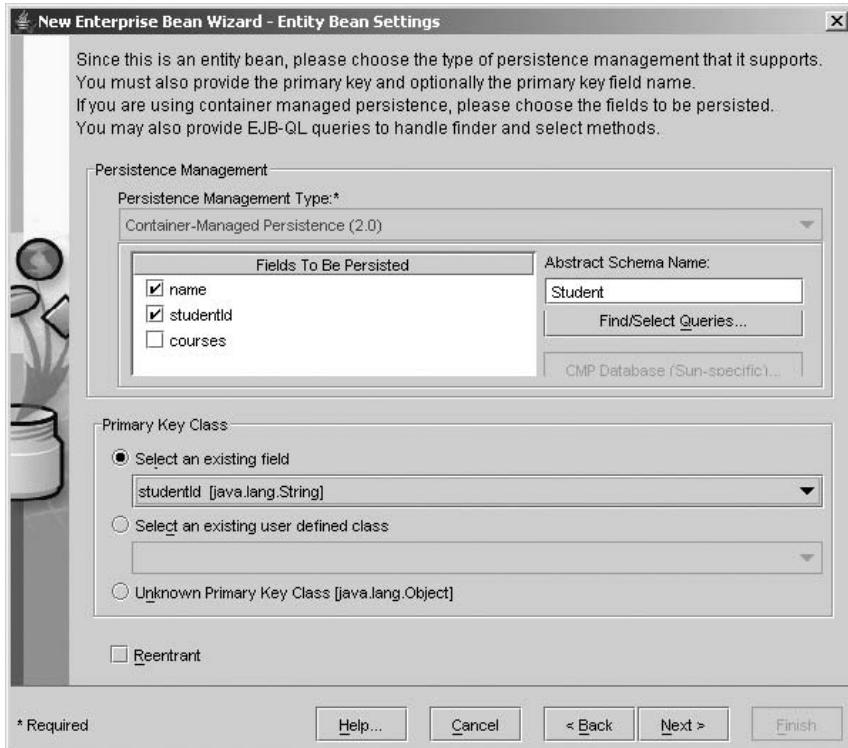
Start deploytool.

1. Create a new Application named EnrollerApp.ear
 - In deploytool, select File | New | Application.
 - Click Browse.
 - In the file chooser, navigate to c:\cop\ejb\cmpenroller.
 - In the File Name field, enter EnrollerApp.ear.
 - Click New Application and OK
2. Create StudentBean Entity Bean
 - Select File | New | Enterprise Bean to start the New Enterprise Bean wizard
 - In the EJB JAR dialog box select the Create New JAR File in the Application radio button. In the combo box, select EnrollerApp.
 - In the JAR Name field, enter CmpJAR.
 - Click Edit
 - In the tree under Available Files, locate the c:\cop\ejb\cmpenroller\build directory. Select cmppack folder and click Add, then click OK.
 - In the EJB JAR General Settings dialog box, select cmppack.StudentBean for the Enterprise Bean Class combo box.
 - Verify that the Enterprise Bean name field is StudentBean.
 - Under Bean Type, select the Entity radio button.
 - In the Local Home Interface combo box, select cmppack.LocalStudentHome.
 - In the Local Interface combo box, select cmppack.LocalStudent.
 - Click Next, Finish.



Selecting the Persistent Fields and Abstract SchemaName:

- Select StudentBean and click Entity tab.
- In the Fields To Be Persisted list, select the fields that will be saved in the database. For Student entity bean select studentId and name.



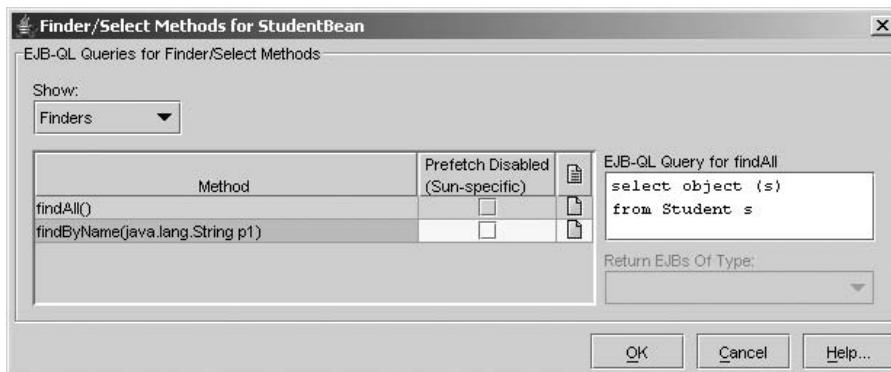
- Choose **Select an existing field** for the Primary Key Class.
- Select **studentId** from the combo box.
- In the Abstract Schema Name field, enter **Student**. This name will be referenced in the EJB QL queries.
Defining EJB QL Queries for Finder and Select Methods:
- Click the **Finder/Select Queries...** button to start **Finder/Select Methods for StudentBean** dialog box.
- To display a set of finder or select methods, click one of the radio buttons under the Show label.
- To specify an EJB QL query, choose the name of the finder or select method from the Method list and then enter the query in the field labeled **EJB QL Query**.
- Select **findAll**, type the following EJB_QL Query for **findAll**:

```
select object(s) from Student s
```

- Select **findByName** and type the following EJB_QL Query for **findByName**:

```
select distinct object(s) from Student s
where s.name = ?1
```

- Click OK.



Specifying Transaction Settings:

- Select **StudentBean** in the tree.
- Click Transactions tab.
- Select Container-Managed.

3. Adding CourseBean Entity Bean to Existing JAR File

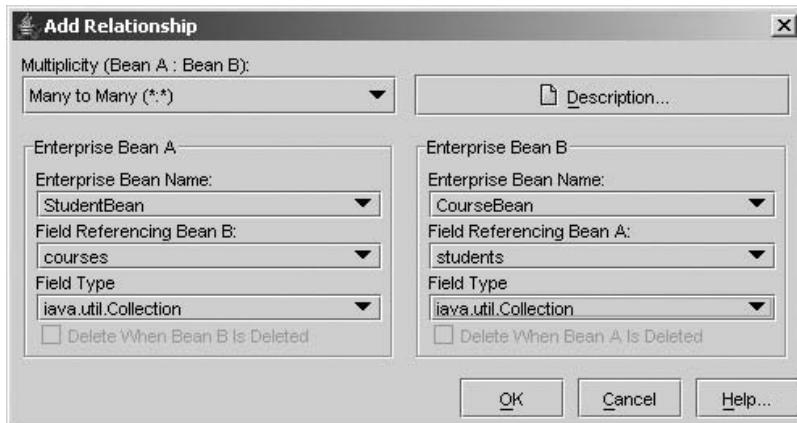
- Select **File | New | Enterprise Bean** to start the New Enterprise Bean wizard.
- In the EJB JAR dialog box, select **Add to Existing JAR File**.

- In the Existing JAR File, select CmpJAR and click Next.
- Follow the steps of creating entity bean of StudentBean to complete the CourseBean creation.

4. Defining Entity Relationships

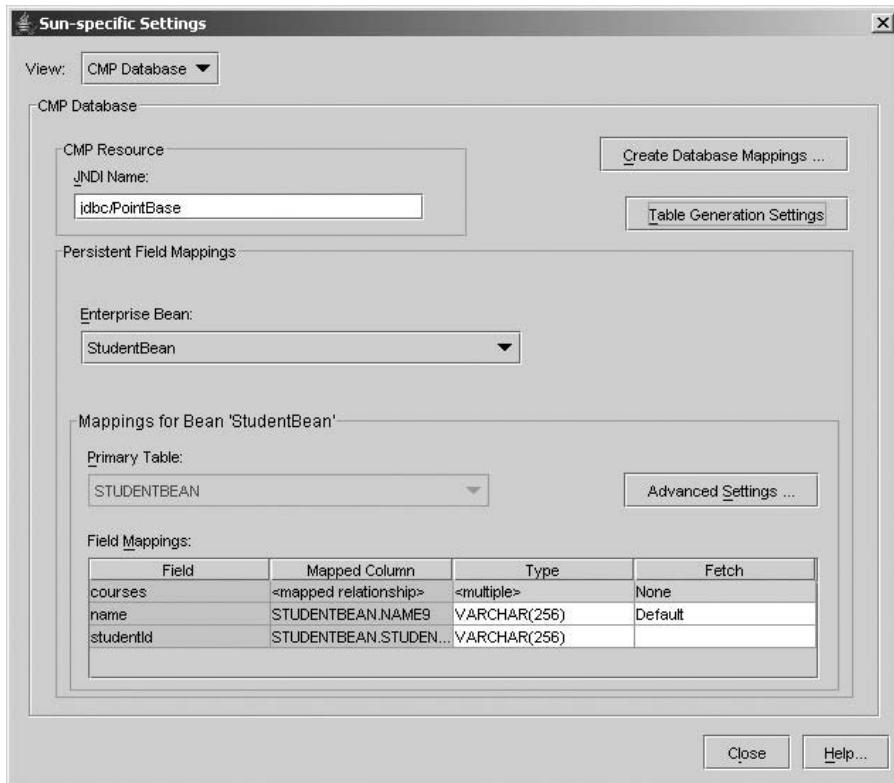
After you create StudentBean and CourseBean you can define relationships between the two entity beans that reside in the same EJB JAR file.

- Select CmpJAR in the tree view and then select the relationships tab.
- Click Add (or Edit to modify) button to open Add Relationship dialog box.
- Select many to many in the combo box
- In the Enterprise Bean A, select StudentBean for Enterprise Bean Name, courses for Field Referencing Bean B, and java.util.Collection for Field Type.
- In the Enterprise Bean B, select CourseBean for Enterprise Bean Name, students for Field Referencing Bean A, and java.util.Collection for Field Type.



5. Specifying Table Creation:

- Select CmpJAR from the tree in deploytool..
- Select the General tab.
- Click Sun-specific Settings button.
- In the JNDI Name field enter jdbc/PointBase.
- Click Create DataBase Mappings button.
- Select Automatically Generate Necessary Tables.
- Click OK.
- Verify field mappings for StudentBean and CourseBean.
- Click the Table Generation Settings to open a dialog box.
- Select Create Table on Deploy and Delete Table on Undeploy for testing and click OK.
- Click Close.



6. Packaging the Session Bean – EnrollerBean

- Select **File | New | Enterprise Bean** to start the New Enterprise Bean wizard.
- Select the **Create New JAR File In Application** radio button.
- In the combo box, select **EnrollerApp**.
- In the **JAR Name** field, enter **EnrollerJAR** and then click **Edit Contents**.
- In the tree under Available Files, locate the **c:\cop\ejb\cmpenroller\build** directory.
- Select the **Enroller** folder from the Available Files tree and click **Add**:
- In the **Bean General Settings** dialog box select **enroller**, **enrollerBean** for **Enterprise Bean Class** combo box.
- Enter **Enrollerbean** in the **Enterprise Bean Name** field.
- Select **Stateful Session** radio button for **Bean type**.
- In the **Remote Home interface** combo box, select **enroller.EnrollerHome**.
- In the **Remote Interface** combo box, select **enroller.Enroller**.
- Click **Next, Finish**.

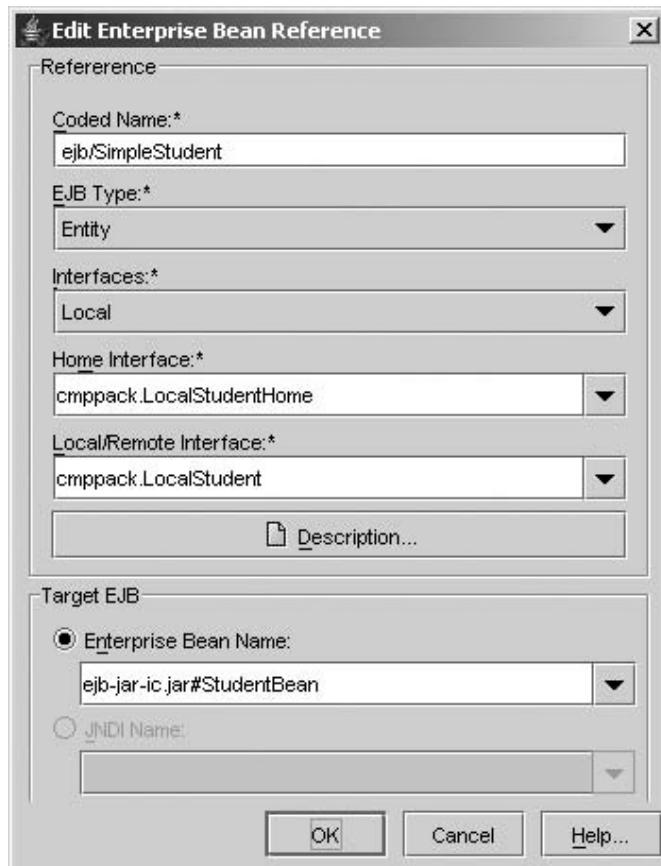
Specify the Session Bean Reference:

The `EnrollerEJB` session bean accesses two entity beans: `StudentBean` and `CourseBean`. When it invokes the lookup method, the `EnrollerBean` refers to the home of entity beans:

```
Object objref =
    initial.lookup("java:comp/env/ejb/SimpleStudent");
Object objref =
    initial.lookup("java:comp/env/ejb/SimpleCourse");
```

Specify this reference as follows:

- In the tree, select `EnrollerBean`.
- Select the EJB Refs tab and click Add.
- In the Coded Name column, enter `ejb/SimpleStudent`.
- In the Type column, select Entity.



- In the Interfaces column, select Local.
 - In the Home Interface column, enter cmppack.LocalStudentHome.
 - In the Local/Remote Interface column, enter cmppack.LocalStudent and click OK.
 - Choose Enterprise Bean Name for Target EJB and select ejb-jar-ic.jar#StudentBean from the combo box.
 - Add another line for CourseBean.
- Specifying Transaction Settings:
- Select EnrollerBean in the tree.
 - Click Transactions tab.
 - Select Container-Managed

7. Packing the Application Client

Create new application client module:

- Select File | New | Application Client to open Application Client wizard.
- In the JAR File Contents dialog box select Create New AppClient Module in Application.
- Select EnrollerApp in the combo box.
- Enter EnrollerClient in the Appclient name field.
- Click Edit Contents.
- In the tree under Available Files, locate the c:\cop\ejb\cmpenroller\build directory.
- Select the Client folder For the Contents.
- Click OK and Next.
- In the General dialog box select client.EnrollerClient for the Main Class combo box.
- In the Callback Handler Class combo box, select container-managed authentication.

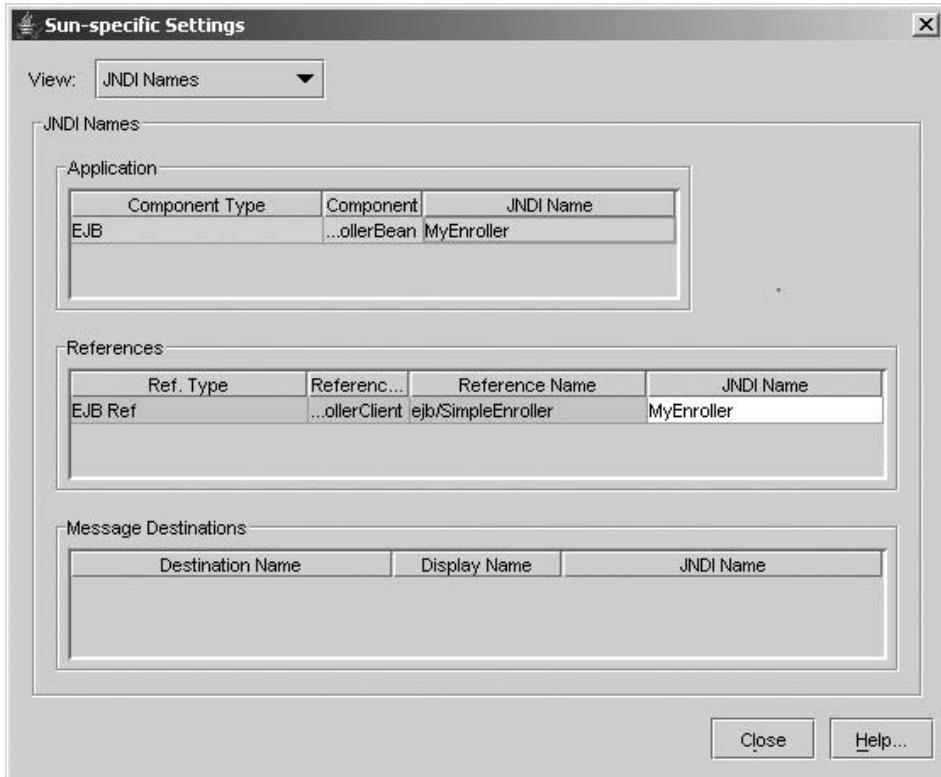
Specifying the ApplicationClient Reference:

- In the tree, select EnrollerClient.
- Select the EJB Refs tab and click Add.
- In the Coded Name field, enter ejb/SimpleEnroller.
- In the Type field, select Session.
- In the Interfaces field, select Remote.
- In the Home Interface field, enter enroller.EnrollerHome.
- In the Local/Remote Interface field, enter enroller.Enroller.
- Select JNDI Name for the Target EJB and enter MyEnroller and click OK.

Specifying the JNDI Names:

- In the tree, select EnrollerApp.
- Click Sun-Specific Settings button.
- In the Application table, locate the EnrollerBean component and enter MyEnroller in the JNDI name field.

- To map the references, in the References table enter MyEnroller in the JNDI name.



8. Final deployment
 - Click Tools | Deploy
 - Select Return Client Jar and click OK.

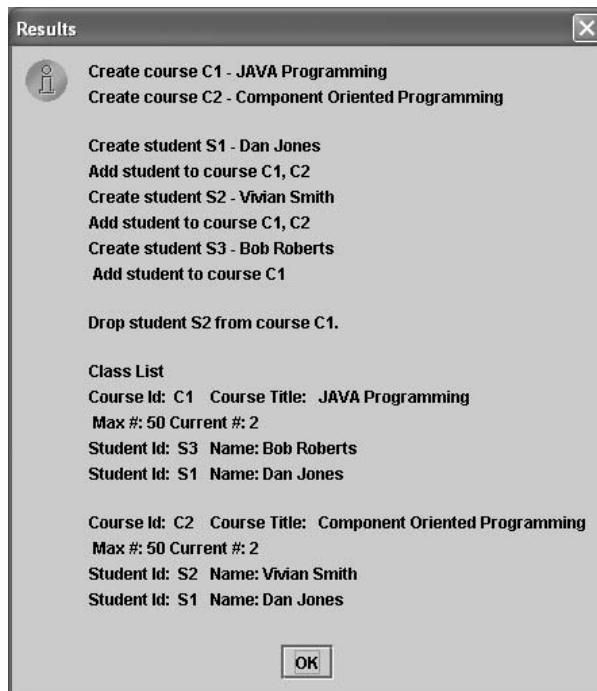
Step 5: Running the EnrollerApp Application

In c:\cop\ejb\cmpenroller directory run the following command:

appclient -client EnrollerAppClient.jar

The report of this example shows that the student registration system has courses C1 and C2 and students S1, S2, and S3. S1 registered in C1 and C2. S3 student registered in C1 and C2 as well. S3 registered in C1. S2 dropped C1.

The final report shows that course C1 – “JAVA Programming” has two students, the course C2 “Component-Oriented programming” has S1 and S2 registered.



4.6 SUMMARY

Sun Microsystems' EJB 2.x provides a complete component infrastructure similar to CORBA 3.0. The component model of the EJB technology is used for building Java enterprise application in distributed computing environment. Any EJB component has two interfaces and one bean class to implement the interfaces. The EJB home (can be local home or remote home) interface manages the life cycle of an EJB object such as creating, removing an EJB instance, and finding an entity bean instance by primary key. The `EJBObject` (remote or local) interface provides and exposes all business logic methods to be used by its clients. The implementation bean class extends a session bean or entity bean class, depending on the type of this EJB component.

There are two types of session beans, stateless and stateful ones, and both of them act on the behaviors of their clients. The stateless EJB component does not hold any state information during its session such as an on-line calculator. On the other hand, stateful session beans need to hold some state information during the session. For instance, a shopping cart session bean needs to keep track of a client's items in a collection.

There are two types of entity beans, too, BMP and CMP ones, and both of them are used in the back-end tier to connect database and to support the persistence. Each entity bean has a relation table behind it. Each instance of the entity EJB component is stored in the table as a row. CMP entity beans are free of SQL database access code, and the mapping to database implementation is done by a deployer at deployment time.

Any EJB component is hosted by an EJB container, which provides all necessary services and makes EJB more portable and scalable. An EJB component is packed, assembled, and finally deployed on a server.

EJB components are server-side reusable components. This compares to JavaBeans components discussed in Chapter 3, which are client-side reusable components. EJB components are widely used in distributed computing applications in a LAN, WAN, Internet, or wireless network environment. The EJB component can work with other components such as CORBA components, Servlets, or JSP Web components.

4.7 SELF-REVIEW QUESTIONS

(True or False)

1. BMP entity beans provide SQL access code but CMP entity beans do not.
2. CMP entity beans can be deployed to any J2EE server with any relational database.
3. The `ejbCreate()` method is in the home interface of an entity bean.
4. Every session bean has a primary key.
5. MDB Java beans also have their interfaces just like ordinary enterprise beans.
6. Web service endpoints can be applied to any entity bean to make an EJB component Web service available.
7. The `ejbCreate()` method is called back by its container.
8. The `ejbCreate()` method must be overridden by an entity bean.
9. Session beans may have their own states.
10. Any EJB component must have a remote interface and a home interface.
11. The business logic is defined in remote interface.
12. The finder method should be defined in both session beans and entity beans.
13. An EJB component can be reached by any Servlet Web component or JSP Web component or HTML component.
14. An EJB component cannot have both of local interface and remote interface.

Keys to Self-Review Questions

1. T
2. T
3. T
4. F
5. F
6. F
7. T
8. T
9. T
10. F
11. T
12. F
13. T
14. F

4.8 EXERCISES

1. What is EJB component infrastructure?
2. How is an EJB component object located?
3. How does the EJB container work?
4. What is the purpose of the EJB container?

5. What are the major differences between EJB and Javabeans?
6. What are the major differences between EJB and MDB?
7. Is EJB architecture a cross-language platform?
8. What are new in EJB 2.x compared with EJB 1.x?
9. What is an object reference?
10. Where will EJB components be deployed?
11. Why do we need local interfaces?
12. What is a local home?
13. What is a local EJB object interface?
14. What is a remote interface?
15. When is a class channel needed?
16. What is EJB packaging?
17. What is EJB assembly?
18. What would a .jar file usually contain?
19. What would a .war file usually contain?
20. What would a .ear file usually contain?

4.9 PROGRAMMING EXERCISES

1. Session bean programming
 - a. Design a stateful session bean that provides a calculator services. The calculator has the functionality to perform addition, subtraction, multiplication, and division of two real numbers. It can also detect the zero divisors in division operation. Use namespace in this component.
 - b. Design a client of the calculator in the previous question in Web component or text-mode client.
 - c. Deploy the server component “Calculator.”
 - d. Plug in this “Calculator” component into a Window Form application.
 - e. Plug in this “Calculator” component into a Web form to be browsed by any Web browser.
 - f. Plug in this “Calculator” component into a text-mode client.
2. Entity bean programming
 - a. Design an on-line auction business application using entity beans and session beans and other Java Web components. The business includes all routine auction activities such as bidding price, starting price, current highest bid price, time period control, winner notification, and so on.

REFERENCES

- [Armstrong 2004] Armstrong, E. *J2EE 1.4 Tutorial*, java.sun.com/j2ee/1.4/docs/tutorials/doc, 2004.
- [Component 2004] www.componentsource.com, 2004.
- [Pawlan 2000] Pawlan, M. *Writing Enterprise Application with Java 2 SDK*, Enterprise Edition, 2000.
- [Sun2 2003] Sun Microsystems, *J2EE Developer's Guide*, java.sun.com, 2003.
- [Sun1 2002] Sun Microsystems, *Enterprise JavaBean Specification, v2.1*, java.sun.com, 2002.

5

CORBA COMPONENTS

Objectives of This Chapter

- Introduce CORBA component infrastructure
- Introduce the concepts of CORBA components and its runtime environment
- Discuss different types of CORBA components, their connections, and deployments
- Introduce new features of CORBA Component Model (CCM)
- Provide step-by-step guide on building, deploying, and using CORBA components

5.1 CORBA COMPONENT INFRASTRUCTURE

5.1.1 CORBA Overview

The Common Object Request Broker Architecture (CORBA) is becoming an industrial standard for component-based software and distributed middleware development. CORBA was created by a consortium called the Object Management Group (OMG), involving over 800 companies. It allows distributed components to interoperate in a language-independent and platform-independent environment. Since CORBA components are self-descriptive, they can be deployed on any compliant servers. A CORBA component is portable and scalable. A client CORBA component can be programmed in a programming language different from the language used for implementing the server component. Besides, a client CORBA component can be deployed and executed on a platform that is different from the platform where the server component is running.

CORBA 1.0 was created in 1991 as a distributed object model. It had an Interface Definition language (IDL), which supports a mapping to C, and a set of API for remote method invocation and interface repository. This earlier version only supported BOA (Basic Object Adapter) not POA (Portable Object Adapter). As a matter of fact, CORBA 1.0 was not a true component infrastructure.

CORBA 2.0 was released in 1996. CORBA 2.2 defined POA, Internet Inter-ORB Protocol (IIOP), and supported the mapping from IDL to Java in addition to C. CORBA 2.x increased the interoperability by specifying a mandatory IIOP, which is basically TCP/IP plus the message format exchanges that serves as a common backbone bus protocol. Any CORBA vendor must implement IIOP so that their CORBA objects could talk to other CORBA objects on the Internet. There are a number of vendors implementing CORBA 2.x such as Borland VisiBroker with IDL to C++ and Java mapping, IONA's Orbix with IDL to C++, Smalltalk, and Java mapping, Sun Microsystems JIDL with IDL-to-Java mapping, IBM SOM (System Object Model), and so on. Microsoft has its own CORBA-compliant product called DCOM, which is being replaced by .NET framework.

CORBA 3.0, which is the latest version at the time of writing this book, is a true component infrastructure because the CCM (CORBA Component Model) became an integral part of CORBA 3.0. More about CCM will be discussed in the next few sections. OpenCCM, MicoCCM, and EJB 3.0 are examples of the implementations for the CCM standard.

The idea of CORBA originated from Remote Procedure Call (RPC) in Unix and C. RPC is procedure oriented and programming language dependent. In other words, RPC does not pass any object as an argument or return an object as the result of calling a remote procedure. CORBA defines the infrastructure for interoperations of distributed component objects. It handles a request for services provided by a distributed component. The services provided by a component are exposed via its interface described by an IDL program. Each object of CORBA-distributed components is identified by its object reference. IDL is a definition language with its own syntax and semantics. IDL also supports a number of data types such as long, short, float, double, char, boolean, and other complex data types.

5.1.2 CORBA Architecture Basics

Let us take a look at all the important parts in CORBA architecture first. Figure 5.1 shows a diagram of CORBA architecture. IIOP is a common backbone bus where CORBA components and their clients are running and communicate via the Object Request Broker (ORB).

1. *Object Request Braher (ORB)* is a container software that has CORBA runtime library and processes to locate and activate any remote object. ORB mediates the interactions between clients and servers and provides distributed services that handle the request to remote CORBA objects. It locates remote objects, requests remote methods in a remote CORBA interface, and gets the result back to the client. ORB must be available in both client and server sides.
2. *Object Adapter (OA)* is a runtime environment that is in charge of CORBA component object's life cycle on the server side. It provides API for generation

and interoperations of OR (Object Reference), method invocation, interaction security, object activation and deactivation, mapping OR to its object's implementation, and so on. OA associates a CORBA component object with ORB.

BOA provides basic adapter services to listen to a client's connection request and redirect the inbound request to the desired target object. However, the object is not portable in BOA.

POA is replacing BOA because it allows CORBA components to be portable between different ORB vendors. POA also supports CORBA objects with persistent identities and transparent activation of objects.

3. *Stub and Skeleton* The stub code and the skeleton code are used to marshal and demarshal remote method invocations in distributed applications. Marshaling is an encoding process to pack all information about remote method invocation in a flat format to be sent to the remote destination, while demarshaling is an opposite process to unpack and decode the messages. The stub marshals the method invocation request and the skeleton demarshals the request and forwards it to an actual remote method invocation. Both of them make a CORBA component and its client be aware of the definition of IDL, that is, the definition of the interface for this CORBA component. There are two different approaches to generating the stub code for a client and generating the skeleton code for the server component: (1) static approach, SII (Static Invocation Interface) and SSI (Static Skeleton Interface); (2) dynamic approach, DII (Dynamic Invocation Interface) and DSII (Dynamic Skeleton Interface). SII and SSI are generated at IDL compilation time. It means that the IDL interface must be available in advance. In some cases, it is not possible to have these available. For example, we need to build a bridge for an adapter object to hook up with an existing CORBA system without rebuilding it. In such a case, the client can get all the necessary knowledge of the interface from IR (Interface Repository) as long as the interface is registered. DII consults with IR to find all the syntax of operations to generate and revoke the stub code dynamically. DSII also provides a runtime building mechanism to generate the skeleton code handling the incoming request. It does not rely on the static skeleton since it is not available at this time.
4. *Interface Repository (IR) and Implementation Repository* IR is a database on the server side, which has all metadata of all registered IDL interfaces, including type information, methods, and parameters. IR provides self-descriptive binary interfaces. IR is referenced by both DII and DSII. The implementation repository is a database on the server side that tells how to launch a server component when it is not active. It is a runtime repository for all the information about the server components, including classes and objects.
5. *Object References (OR)* are CORBA component objects. OR encapsulate the locations of CORBA object requests and other information such as the IDs of objects. They are proxy objects standing on the client side. They can be passed from one object to another. A client must get the OR to the CORBA component objects in order to invoke a remote method of a CORBA component object.
6. *Interface Definition Language (IDL) and Language Mapping* The IDL interface of a CORBA component exposes all the operations or methods that a client of this component may access. An IDL interface is a language-independent text file with an extension `.idl`, which can be mapped to many different programming languages such as C++, Java, Smalltalk, and so on. The details of IDL will be

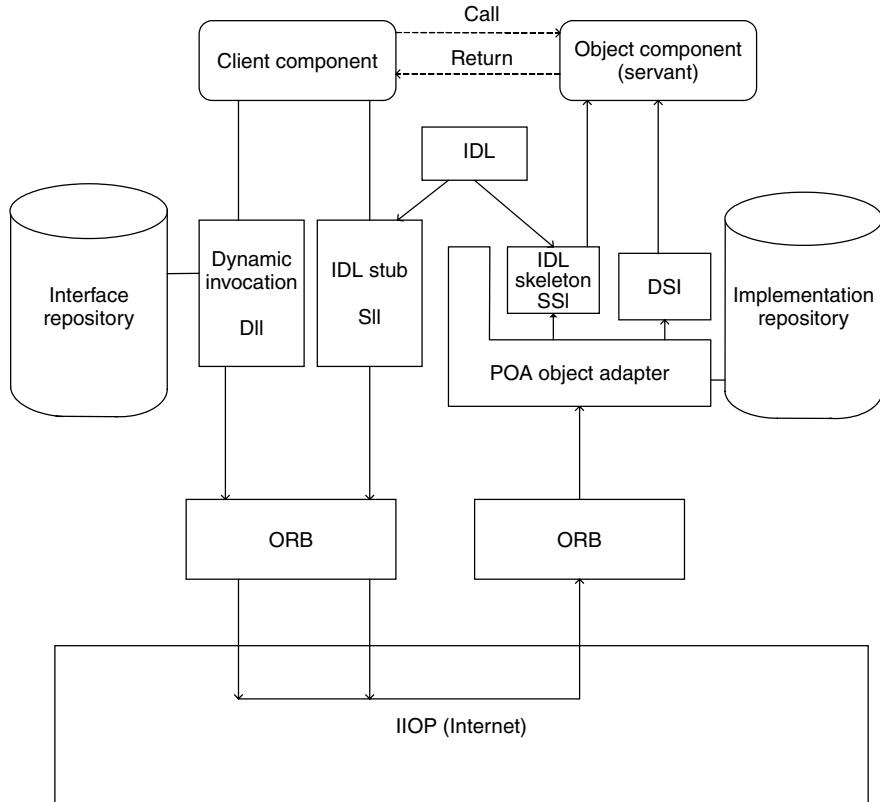


FIGURE 5.1. CORBA infrastructure.

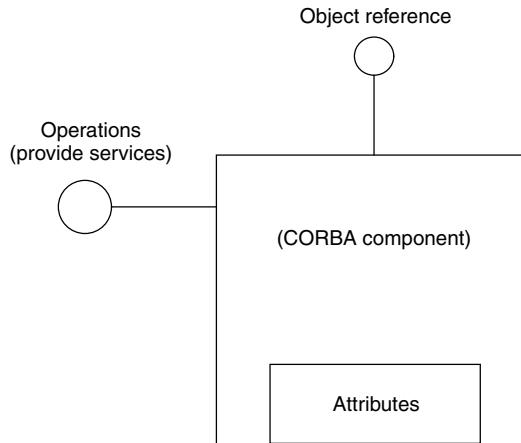
discussed in next section. Just like any other component architecture, the IDL interface is a contract between a server component and its client that separates the implementation from the definition.

The dash lines in Figure 5.1 indicate the logical communication channels between a CORBA component object and its client. The solid lines indicate the physical channels for a client to request a service from a server-side CORBA component [Component 2004].

5.2 CORBA COMPONENT MODEL (CCM)

5.2.1 Classic CORBA Object

As we discussed before, the IDL interface of a CORBA component exposes all operations it provides and all attributes that clients can use *get* and *set* methods to access them. The IDL interface is implemented by classes of CORBA component and is aware to any client of the CORBA component. It is a contract between a client and the server component. The basic structure of a CORBA component object is described in Figure 5.2.

**FIGURE 5.2.** CORBA component.

We may have an `Account.idl` as follows:

```

module MyAccount {
    interface Account{
        attribute long accountNo;
        void deposit (in double amount);
        void withdraw(in double amount);
        double reportBalance(); }
}
  
```

The module `MyAccount` is mapped to a package `MyAccount` in Java and the interface `Account` is mapped to a Java interface `Account`. The read/write attribute `accountNo` is mapped to a pair of `get` and `set` methods for this attribute.

Let us take a look at a simple component `TempConvert`, which provides services to convert a temperature from Celsius to Fahrenheit. It is a programming language-independent interface shown below. We can map this IDL interface to any programming language that can be implemented by either the server component or its client. The processing is shown in Figure 5.3.

The server component and its client can be implemented by different languages and running at different platforms. Here we focus on Java implementation.

The first step is to define an interface by IDL in a `Convert.idl` file.

```

module TempConvertApp
{
    interface Convert
    {
        double cToF(in double c);
    };
}
  
```

The utility command `idlj` maps this IDL interface to Java and generates number of files including CORBA interface, client stub, server skeleton, and other helper Java files.

Let us discuss these generated Java files first because they are the bases of the server implementation and the client development. There are two server-side mapping models. One is the inheritance model and the other is the delegation model. We present here a POA inheritance model.

- `Convert.java` specifies the Java interface for the component. We can find that the module is mapped to a package and the interface is mapped to a Java interface. If the IDL interface is mapped to C++, then the module is mapped to a namespace and the interface is mapped to an abstract class in C++. It depends on the vendor's implementations. Figure 5.3 depicts a classic CORBA object development, its structure, and its generation.

```
Package TempConvertApp
Public interface Convert extends org.omg.CORBA.Object{
    Double cToF(double c); }
```

- `TempConvertApp/ConvertOperations.java` declares all exposed operations in this interface.

```
package TempConvertApp;
public interface ConvertOperations
{
    double cToF (double c);
} // interface ConvertOperations
```

- `TempConvertApp/ConvertHelper.java` provides auxiliary functionality such as the `narrow()` method required to cast a CORBA object reference to their proper types.

```
package TempConvertApp;

abstract public class ConvertHelper
{
    private static String _id =
        "IDL:TempConvertApp/Convert:1.0";

    public static void insert (org.omg.CORBA.Any a,
                           TempConvertApp.Convert that)
    {
        org.omg.CORBA.portable.OutputStream out =
            a.create_output_stream();
        a.type (type ());
        write (out, that);
        a.read_value (out.create_input_stream (), type ());
    }

    public static TempConvertApp.Convert extract (org.omg.CORBA.Any a)
    {
        return read (a.create_input_stream ());
    }

    private static org.omg.CORBA.TypeCode __typeCode = null;
    synchronized public static org.omg.CORBA.TypeCode type ()
```

```

{
    if (_typeCode == null)
    {
        _typeCode = org.omg.CORBA.ORB.init ().create_interface_tc
                    (TempConvertApp.ConvertHelper.id (), "Convert");
    }
    return _typeCode;
}

public static String id ()
{
    return _id;
}

public static TempConvertApp.Convert read
(org.omg.CORBA.portable.InputStream istream)
{
    return narrow (istream.read_Object (_ConvertStub.class));
}

public static void write (org.omg.CORBA.portable.OutputStream
                        ostream,
TempConvertApp.Convert value)
{
    ostream.write_Object ((org.omg.CORBA.Object) value);
}

public static TempConvertApp.Convert narrow
(org.omg.CORBA.Object obj)
{
    if (obj == null)
        return null;
    else if (obj instanceof TempConvertApp.Convert)
        return (TempConvertApp.Convert)obj;
    else if (!obj._is_a (id ()))
        throw new org.omg.CORBA.BAD_PARAM ();
    else
    {
        org.omg.CORBA.portable.Delegate delegate =
        ((org.omg.CORBA.portable.ObjectImpl)obj)._get_delegate ();
        TempConvertApp._ConvertStub stub = new
        TempConvertApp._ConvertStub ();
        stub._set_delegate(delegate);
        return stub;
    }
}
}

```

- `TempConvertApp/ConvertHolder.java` has a Holder class used to hold a CORBA object for input stream read and output stream write operations of parameters.

```

package TempConvertApp;
public final class ConvertHolder implements

```

```

    org.omg.CORBA.portable.Streamable
{
    public TempConvertApp.Convert value = null;

    public ConvertHolder ()
    {
    }
    public ConvertHolder (TempConvertApp.Convert initialValue)
    {
        value = initialValue;
    }
    public void _read (org.omg.CORBA.portable.InputStream i)
    {
        value = TempConvertApp.ConvertHelper.read (i);
    }

    public void _write (org.omg.CORBA.portable.OutputStream o)
    {
        TempConvertApp.ConvertHelper.write (o, value);
    }
    public org.omg.CORBA.TypeCode _type ()
    {
        return TempConvertApp.ConvertHelper.type ();
    }
}

```

- TempConvertApp/ConvertPOA.java is a skeleton class for server implementation that implements operations interface and uses narrow() in Helper class shown before. This is a stream-base skeleton.

```

package TempConvertApp;
public abstract class ConvertPOA extends
    org.omg.PortableServer.Servant
implements TempConvertApp.ConvertOperations,
    org.omg.CORBA.portable.InvokeHandler
{
    // Constructors
    private static java.util.Hashtable _methods = new
        java.util.Hashtable ();
    static
    {
        _methods.put ("cToF", new java.lang.Integer (0));
    }
    public org.omg.CORBA.portable.OutputStream _invoke (String
        $method, org.omg.CORBA.portable.InputStream in,
        org.omg.CORBA.portable.ResponseHandler $rh)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        java.lang.Integer __method=(java.lang.Integer)_methods.get
            ($method);
        if (__method == null)
            throw new org.omg.CORBA.BAD_OPERATION (0,
                org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
    }
}

```

```

// Dispatch method request to its handler
switch (_method.intValue ())
{
    case 0: // TempConvertApp/Convert/cToF
    {
        double c = in.read_double ();
        double $result = (double)0;
        //invoke the method
        $result = this.cToF (c);
        //create an output stream for delivery of the result
        out = $rh.createReply();
        //Marshal the result via output stream which connects
        //the input stream of client
        out.write_double ($result);
        break;
    }
    default:
        throw new org.omg.CORBA.BAD_OPERATION (0,
            org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
}
return out;
} // _invoke

// Type-specific CORBA::Object operations
private static String[] __ids = {
    "IDL:TempConvertApp/Convert:1.0"};

public String[] _all_interfaces
    (org.omg.PortableServer.POA poa, byte[] objectId)
{
    return (String[])__ids.clone ();
}

public Convert _this()
{
    return ConvertHelper.narrow(
        super._this_object());
}
public Convert _this(org.omg.CORBA.ORB orb)
{
    return ConvertHelper.narrow(
        super._this_object(orb));
}
} // class ConvertPOA

```

- `TempConvertApp/ConvertStub.java` is a stub for a CORBA client that marshals the arguments of method invocation via an output stream and unmarshals the results back via an input stream. The stub class also implements the `Convert` Java interface shown before.

```

package TempConvertApp;
public class _ConvertStub extends
    org.omg.CORBA.portable.ObjectImpl

```

```

implements TempConvertApp.Convert
{
    public double cToF (double c)
    {
        org.omg.CORBA.portable.InputStream $in = null;
        try {
            //create a request via an output stream
            org.omg.CORBA.portable.OutputStream $out =
                _request ("cToF", true);
            //marshal the arguments
            $out.write_double (c);
            //method invocation via output stream and
            //connect to a input stream
            $in = _invoke ($out);
            //unmarshal the return result
            double $result = $in.read_double ();
            return $result;
        } catch (org.omg.CORBA.portable.ApplicationException $ex) {
            $in = $ex.getInputStream ();
            String _id = $ex.getId ();
            throw new org.omg.CORBA.MARSHAL (_id);
        } catch (org.omg.CORBA.portable.RemarshalException $rm) {
            return cToF (c);
        } finally {
            _releaseReply ($in);
        }
    } // cToF
    // Type-specific CORBA::Object operations
    private static String[] __ids =
        {"IDL:TempConvertApp/Convert:1.0"};

    public String[] __ids ()
    {
        return (String[])__ids.clone ();
    }

    private void readObject (java.io.ObjectInputStream s) throws
        java.io.IOException
    {
        String str = s.readUTF ();
        String[] args = null;
        java.util.Properties props = null;
        org.omg.CORBA.Object obj = org.omg.CORBA.ORB.init (args,
            props).string_to_object (str);
        org.omg.CORBA.portable.Delegate delegate =
            ((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate
            ();
        _set_delegate (delegate);
    }

    private void writeObject (java.io.ObjectOutputStream s)
        throws java.io.IOException
    {
        String[] args = null;
        java.util.Properties props = null;

```

```

        String str = org.omg.CORBA.ORB.init (args,
            props).object_to_string (this);
        s.writeUTF (str);
    }
} // class _ConvertStub

```

- ConvertServer.java is a server file with two classes. One is ConvertImpl class that inherits ConvertPOA class (a CORBA skeleton) and the other is a public daemon class ConvertServer that has a main() method.

```

// ConvertServer.java
import TempConvertApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.util.Properties;

class ConvertImpl extends ConvertPOA {
    private ORB orb;

    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    // implement cToF() method
    public double cToF(double c) {
        return (c*9./5+32);
    }
}

public class ConvertServer {

    public static void main(String args[]) {
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);
            // get reference to rootpoa & activate the POAManager
            POA rootpoa =
                POAHelper.narrow(orb.resolve_initial_references
                    ("RootPOA"));
            rootpoa.the_POAManager().activate();
            // create servant and register it with the ORB
            ConvertImpl convertImpl = new ConvertImpl();
            convertImpl.setORB(orb);
            //get object reference from servant
            org.omg.CORBA.Object ref =
                rootpoa.servant_to_reference(convertImpl);
            Convert href = ConvertHelper.narrow(ref);
            //get naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");

```

```
// cast the generic object reference to a proper type
NamingContextExt ncRef =
NamingContextExtHelper.narrow(objRef);
//bind the name "Convert" with naming service
NameComponent path[] = ncRef.to_name("Convert");
ncRef.rebind(path, href);
// wait for invocations from client
orb.run();
}
catch (Exception ex) {
    System.err.println("ERROR: " + ex);
    ex.printStackTrace(System.out);
}
```

- `ConvertClient.java` is a CORBA GUI client that accesses the CORBA component on the server.

```
import TempConvertApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class ConvertClient extends JFrame
{
    static Convert convertImpl;
    static JTextField input;
    static JTextField output;
    static ConvertClient a;
    static JButton submit, clear;

    public ConvertClient()
    {           //layout the GUI
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JLabel l1=new JLabel("C input:");
        JLabel l2=new JLabel("F output");
        input =new JTextField(10);
        output= new JTextField(10);
        submit=new JButton("SUBMIT");
        clear=new JButton("Clear");
        submit.addActionListener(new ActionHandler());
        clear.addActionListener(new ActionHandler());
        contentPane.add(l1);
        contentPane.add(input);
        contentPane.add(l2);
        contentPane.add(output);
        contentPane.add (submit);
```

```

        contentPane.add(clear);
        setTitle("Client Access");
        setSize(340, 250);
        show();
    }
    public static void main(String args[])
    {
        try{
            a =new ConvertClient();
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);
            // get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            // Use NamingContextExt instead of NamingContext.
            // This is part of the Interoperable naming Service.
            NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef)
            // resolve the Object Reference in Naming
            String name = "Convert";
            convertImpl = ConvertHelper.narrow(ncRef.resolve_str(name));
            } catch (Exception e) {
                System.out.println("ERROR : " + e) ;
                e.printStackTrace(System.out);}
        }
        class ActionHandler implements ActionListener{
        public void actionPerformed(ActionEvent e)
        {
            try{
                if(e.getSource()==submit)
                {
                    System.out.println("Obtained a handle on server
                        object:");
                    String temp=input.getText();
                    double a =Double.parseDouble(temp);
                    double result=convertImpl.cToF(a);
                    output.setText(""+result);
                }
                else if(e.getSource() == clear)
                { input.setText("");
                    output.setText("");
                }
                } catch (Exception ex) {
                    System.out.println("ERROR : " + e) ;
                    ex.printStackTrace(System.out);
                }
            }
        }
    }
}

```

Figure 5.3 shows the CORBA component development guideline. The detail steps to build and to run this application with the CORBA component will be shown in the practice lab section of this chapter.

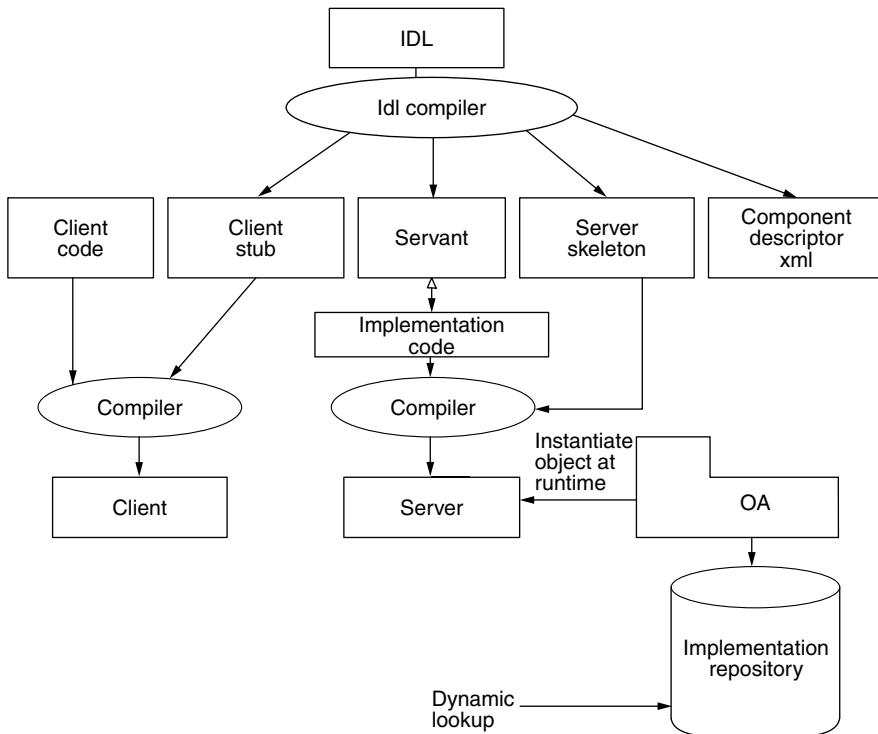


FIGURE 5.3. CORBA component developments.

5.2.2 Component Model of CCM in CORBA 3.x

5.2.2.1 CCM Component Concepts There are two CORBA component types supported in CORBA 3.0. One is the CCM component (extended component) that supports many new component features. The other discussed in last section is a basic classic component that does not support any of the following except the component attribute. The declaration of an extended CCM component type is declared in CIDL (Component Interface Definition Language) and is mapped (translated) into its equivalent IDL declaration [OMG CCM 2002].

CCM has significantly reduced the software complexity, increased the reusability and productivity of software, made it easier to adapt and maintain, and made it easier to extend or aggregate software. CCM also provides a standard facility to support component packaging, assembling, and deployment. A CCM component must be packaged and deployed in order to work and is completely different from classic CORBA object. A CCM component is declared by a keyword *component* in CIDL. Figure 5.4 shows an example of CCM component. The CCM component syntax is as follows [OMG CCM 2002; Flissi 2002].

```

component <name>
[ :<base>][supports <interface>], [,<interface>]
{<attribute declaration>
<port declaration>
}
  
```

A CCM component can inherit from one parent CCM component specified in `<base>` and/or supports multiple interfaces specified in `<interface>`, which is similar to Java class that can extend one parent class and implement many interfaces. In order to get the services provided by a CCM component, a client must create an instance of that component. Each component instance is created and managed by its component home interface. Each component has many ports to offer services or send events to other components and get services or events from other components.

The component-distinguished interface shown in Figure 5.4 is also called an *equivalent interface*. A component instance is identified by its component reference or by its set of facet reference. One component has a single distinguished reference that the interface conforms to the component definition. A component interface is an entry point that is used to navigate other multiple interfaces supported by this component. This interface allows the client to navigate to the component's facets and from any facet to the component interface by `get_Component()` method. The facet interfaces are encapsulated by the component. Any component instance is created by its home interface and accessed via an object reference. The home interface and object reference are discussed later in this section. The type of this object reference is an interface, and it is not explicitly defined and is called equivalent interface or distinguished interface. This interface itself does not define any method. All the methods encapsulated are derived from other supported interfaces. A component definition cannot introduce any new operations. The only way to expose operations in a component is to include these methods in a supported interface.

A CCM component binds its features to its ports. The CCM ports make up a component capacity. The loose coupling features of CCM ports make it much easier to compose a new component.

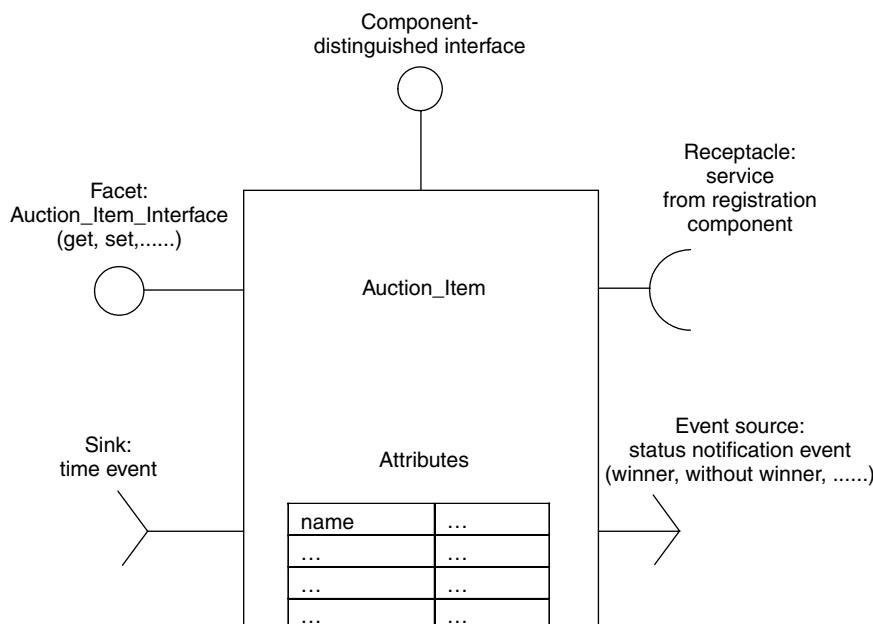


FIGURE 5.4. CCM component and its ports.

The ports supported by CCM component can be one of the following four:

1. *Facets* The facet is a CCM port of access point or entry point. A CCM component provides and exposes all its functionality as a set of interfaces that come from supported interfaces by this component. All interfaces provided by a component are called the component's facets. One component may provide many facets. In Figure 5.4, you can find an *auction_item* component that has a facet named *auction_Item_Interface*. All the operations supported by this *auction_item* such as setting the starting bid price, setting the new bid price, getting the current bid price, and others are exposed via this facet so that the clients of this component can operate on it. A facet of a CCM component is implemented by its component executor. The syntax of a facet definition is

```
provides <Interface> <name_identifier>
```

Its equivalence in CORBA 2.x IDL is:

```
<interface> provide_<name_identifier>()
```

A facet name is used to navigate between component equivalent interface and individual facets. We can navigate from equivalent reference to its facets by a method `provide_facet(facet_name)` or navigate from a facet to its own component by `get_component()`.

2. *Receptacles* The receptacle is a connection point to collaborate with other CCM ports. It is a kind of configuration port to specify some required services from other components. A CCM component needs to perform some operations provided by other components to complete its own task. A CCM component must obtain an object reference to the instance of other components in order to get services from other components. A CCM component uses the receptacle to connect to other components to receive services. The *auction_item* component gets the services from another component called *registration*, which provides services to register or unregister an auction item to or from the system.

The syntax of a receptacle definition is

```
uses <Interface> <name_identifier>
```

It is mapped to its IDL equivalence connecting, disconnecting, and getting related reference as below:

//receptacle passes an object reference to make a connection

```
void connect_<name_identifier>(in <interface> conxn);

//disconnection

<interface> disconnect_<name identifier>();

//returns the connected object reference if it is connected.

<interface> get_connection_<name identifier>
```

3. *Event Source* OpenCCM supports event-driven communication, which is different from invocation-based in that the event-driven connection is a loosely coupling connection between components. It is also called *asynchronous message passing*. All notification event values must be defined in `ValueType`.

A user-defined event type can be defined as

```
ValueType <event_name>:component::EventBase{
    Public <type> <name_identifier> }
```

A CCM component can send out an event that has an `eventType` to notify its listener who is interested in such an event. A listener is called an *event sink*. The event sent out by a CCM component will be intercepted by any component that has registered and subscribed itself with such an event. We will discuss it in detail in the sink port section. There are two different ways for an event source to communicate with its targets: publish or emit. The publisher mode works in a way for a publisher to broadcast an event to many sinks. It is a one-to-many multiplex mode. The emit mode works in a way of point-to-point mode that an emitter sends an event to one target sink. An emitter can share the same channel such as a message queue with another emitter but a publisher cannot share a channel with another publisher. The `auction_item` component in this example publishes an event of an auction bid status (“sold” or “bid canceled”) to all the bidders. The syntax of an event source can be either one of these two. We can see that an event consumer must subscribe a publisher event source to receive an event and must make a direct connection to an emitter to get an event from it.

```
publishes <event_type> <name_identifier>
```

is mapped to its IDL equivalence as follows:

```
Components::Cookie subscribe_<name_identifier>
    (in <event_type>Consumer consumer);
<event_type>Consumer unsubscribe_<name_identifier>
    (in Components::Cookie ck);
emits   <event_type> <name_identifier>
```

is mapped to its IDL equivalence as follows:

```
Void connect_<name_identifier>(in <event_type>Consumer
    consumer);
<event_type>Consumer disconnect_<name_identifier>();
```

An event-subscribing method returns a cookie to identify the consumer for subsequent unsubscribe operation.

4. *Event Sink* An event destination is called *event sink*, which is interested in receiving this type of event. An event sink must subscribe the event in which it is interested and is expected to be notified. A sink consumes an event from an event source where this component is interested. It is asynchronously notified by an event source, which is very similar to the Java event delegation model where an event listener listens to an event if it is registered with that event.

The syntax for a sink is

```
consumes <event type> <name_identifier>
```

It is mapped to its IDL equivalence, which returns an object reference for the consumer who receives the event.

```
<event type>Consumer get_consumer_<name_identifier>
```

The sink of component `Auction_item` in our example may take a timer event sent out by another clock component when the auction deadline is passed. A coffeemaker is another example that provides users with an interface as a facet, and the power is provided via a power socket. It has a sink to take some incoming event such as timer or temperature control to stop heating and an event from light sensor component to start the coffeemaker. When coffee is ready, an event signal will be sent out in a format of beep or a message to notify the user or another component such as toaster to start making a toast. A sink port of a component is implemented by its component executor.

Other CCM component features:

1. *Attribute* A CCM component attribute is a named configurable component property, which is only intended to be configured and customized as needed. An attribute is intended to be configured at the time of component installation, packaging, assembling, and deployment that makes a component more adaptable. It can be supported by a visual property tool just like Java BDK or Bean Builder. It can also raise an exception when `get` and `set` methods are called. An attribute can be persistent or transient, and it can also be read/write or read-only.

```
attribute <type> <name_identifier>
```

This will generate accessor and mutator operations in the component interface. These operations may throw exceptions by configurations of `getRaises` and `setRaises` subclauses in the attribute declaration.

2. *Home Interface* All components have their *home* interfaces, which manage the life cycle of the component instance including the instantiation and removing an instance of a component type. A client never creates an instance of a CCM component type by itself. A home interface acts like a factory for component instance. A home interface may also have a finder that provides `find_by_primary_key` and `get_primary_key` operations in its IDL equivalence. A home must be declared for every component. Each component instance is managed by a home object in its container at runtime. Home is a primary entry point for a client to get into a component.

A simple home is declared as

```
Valuetype itemKey:Components::PrimaryKeybase{
    Public String name; }
Home Auction_itemHome manages Auction_item primaryKey itemKey{
    Factory CreateAuction_item(in String name);
    Finder FinderAuction_item(in String name);
}
```

An entity CCM component needs a primary key but service components, process components, and session components do not need it.

3. *Component Categories* CCM supports four component categories:

Service CCM component is a stateless session component without key.

Session CCM component is a stateful conversational component without key.

Process CCM component is a keyless (internal key only) durable entity component.

Entity CCM component is a persistent durable keyful component.

The component categories are declared by CIDL, which is an extension to IDL used after a CCM component is declared. A simple example of CIDL is shown here:

```
Composition entity Auction_itemImpl{
    Implement Auction_item;
    home executor Auction_itemHomeImpl delegatesTo abstract
        storagehome
    Auction_itemStorageHome;};
```

4. *Component Persistent State* A component such as `Auction_item` may keep its state from session to session. Its state may persist. This can be described by Persistent State Definition Language (PSDL), which is a subset of CIDL. For example, we can create a storage type `Auction_itemState` as follows.

```
abstract storage Auction_itemState{
state String name;
state float start_price;
state float current_price;};

storagetype PortableAuction_itemState implements
    Auction_itemState{};
```

5. *Home Executor and Component Executors* A component home interface is implemented by its home executor, for example, a programming artifact, which manages component executor and binds to a storage type described by PSDL. A component is implemented by its component executor. Both the home executor and component executor are specified by CIDL. A complete `Auction_item` CCM component declaration is given after the introduction to all basic structures of CCM component.

```
interface item{
    string name();
    float start_price();
    float setCurrent_price();}

component Auction_item supports item{

    provides itemService myItemService;
    uses registrationService myItemregistration
    consumes TimeEvent from_Registration;
    publishes statusInfo itemInfo1;
    emits statusInfo itemInfo2;
```

```

attribute String name;
attribute float start_price;

}

```

5.2.2.2 CCM Implementation: OpenCCM We will see a complete and tested OpenCCM example in this section. This is a simple auction bid process example. There are Client and Server components in the application myDemo. The Client component gets service from Server component synchronously by connecting its receptacle port to Server's facet port. The Client also connects its sink port to Server's event source port so that it can be notified by Server asynchronously when time is out. The Server publishes an event and Client must register with it if it is interested. All components are created by a simple component home managers with primary keys. The component homes are used to instantiate and manage the components. Figure 5.5 shows the relationship between all CCM components in this example.

A client can be a seller to sell an item by setting up a start bid price and a time period for bidders to bid. A client can also be a bidder who bids the item by offering a price. The winner will be the one who makes the highest offer within the time limit. If a bid is lower than the starting price set by the seller, then the *Server* will inform the clients that the bid is too low. The *Server* takes the seller's request and takes the bids from bidders and notifies all participants about the status of the processing. This sample application has been tested successfully in OpenCCM 0.7.

OMG CORBA 3.0 CCM *Server* and *Client* component definition in Figure 5.5 are given as follows [Flissi 2002; OpenCCM 2003]:

```

//Importation of OMG IDL scopes
import Components;

//replace #pragma
typeprefix demo "ccm.objectweb.org";

module myDemo{

//The Service interface provided by the Server component and used
//by its Client components

```

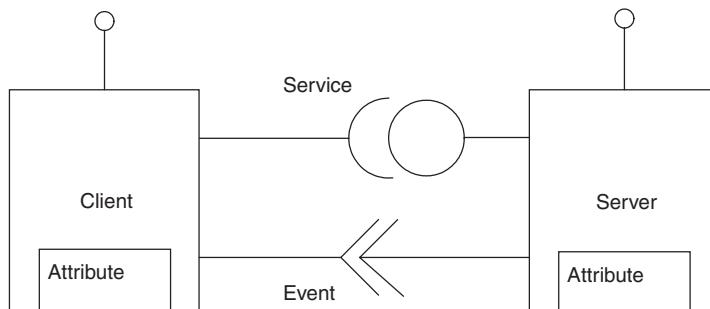


FIGURE 5.5. Relationship between CCM components.

```

        interface MyService
        {
            void show (in string text, in string num);
        };

//The Event valuetype published by the Server component and
//consumed by its Client components
        eventtype TextEvent
        {
            public string text;
        };

//The primary key to identify components
        valuetype NamePrimaryKey : ::Components::PrimaryKeyBase
        {
            public string name;
        };

// The Server component type
        component Server supports MyService
        {
            // Provides a Service to its Client components
            provides MyService my_service;

            // Publishes Events to its Client components
            publishes TextEvent to_clients;

            // attributes can be configured at deployment time.
            attribute string name;
        };

// Simple home for instantiating Server component

        home ServerHome manages Server {};

// The home for managing Server components
        home ServerManager manages Server
            primarykey NamePrimaryKey
        {
            // To create a new Server identified by the name
            factory create_server(in string name);

            // To find a Server identified by the name
            finder find_server(in string name);
        };

// The Client component type
        component Client
        {
            // Uses the service provided by the Server component
            uses MyService my_service;

```

```

// Consumes Events published by Server components.
consumes TextEvent from_servers;

attribute string name;
};

// Simple home for instantiating Client components
home ClientHome manages Client
{
};

// The home for managing Client components
home ClientManager manages Client
primarykey NamePrimaryKey
{
    // To create a new Client identified by the name.
    factory create_client(in string name);

    //To find a Client identified by the name.
    finder find_client(in string name);
};
}

```

The CCM component IDL definitions are in `myDemo.idl3`, which is compiled into `myDemo.idl` to get an equivalent IDL definition by the `ir3_idl2` OpenCCM command. The detail steps of the compilation, packaging, assembling, and deployment are shown in the practice lab section.

An equivalent IDL file is generated by the following mapping.

```

eventtype TextEvent
{
    public string text;
};

//is mapped to idl2 as follows:
valuetype TextEvent : ::Components::EventBase
{
    public string text;
};

interface TextEventConsumer : ::Components::EventConsumerBase
{
    void push_TextEvent(in ::myDemo::TextEvent the_textevent);
};

component Server supports MyService
{
    provides MyService my_service;
    publishes TextEvent to_clients;
        attribute string name;
};

//is mapped to idl2 as follows:

```

```

interface Server : ::Components::CCMObject,
                  ::myDemo::MyService
{
    ::myDemo::MyService provide_my_service();
    ::Components::Cookie subscribe_to_clients(in
        ::myDemo::TextEventConsumer consumer);
    ::myDemo::TextEventConsumer unsubscribe_to_clients(in
        ::Components::Cookie ck);
    attribute string name;
};

home ServerHome manages Server {};
//is mapped to id12 as follows:
interface ServerHomeExplicit : ::Components::CCMHome
{
};
interface ServerHomeImplicit : ::Components::KeylessCCMHome
{
    ::myDemo::Server create();
};
interface ServerHome : ::myDemo::ServerHomeExplicit,
                     ::myDemo::ServerHomeImplicit
{
};

home ServerManager manages Server
    primarykey NamePrimaryKey
{
    factory create_server(in string name);
    finder find_server(in string name);
};

//is mapped to id12 as follows:
interface ServerManagerExplicit : ::Components::CCMHome
{
    ::myDemo::Server create_server(in string name);
    ::myDemo::Server find_server(in string name);
};
interface ServerManagerImplicit
{
    ::myDemo::Server create(in ::myDemo::NamePrimaryKey key);
    ::myDemo::Server find_by_primary_key(in
        ::myDemo::NamePrimaryKey key);
    void remove(in ::myDemo::NamePrimaryKey key);
    ::myDemo::NamePrimaryKey get_primary_key(in
        ::myDemo::Server comp);
};

interface ServerManager : ::myDemo::ServerManagerExplicit,
                        ::myDemo::ServerManagerImplicit { };
component Client
{
    uses MyService my_service;
    consumes TextEvent from_servers;
}

```

```

        attribute string name;
};

//is mapped to id12 as follows:

interface Client : ::Components::CCMObject
{
    void connect_my_service(in ::myDemo::MyService connexion);
    ::myDemo::MyService disconnect_my_service();
    ::myDemo::MyService get_connection_my_service();
    ::myDemo::TextEventConsumer get_consumer_from_servers();
    attribute string name;
};

// The following are server side executors
// Main component executor interface

local interface CCM_Server_Executor :
    ::Components::EnterpriseComponent, ::myDemo::MyService
{
    attribute string name;
};

// Monolithic component executor interface
local interface CCM_Server : ::myDemo::CCM_Server_Executor
{
    ::myDemo::CCM_MyService get_my_service();
};

local interface CCM_Server_Context
    : ::Components::CCMContext
{
    void push_to_clients(in ::myDemo::TextEvent event);
};

// Main component executor interface
local interface CCM_Client_Executor
    : ::Components::EnterpriseComponent
{
    attribute string name;
};

// Monolithic(single class) component executor interface
local interface CCM_Client
    : ::myDemo::CCM_Client_Executor
{
    void push_from_servers(in ::demo::TextEvent event);
};

// Component-specific context interface.
local interface CCM_Client_Context
    : ::Components::CCMContext
{
    ::myDemo::MyService get_connection_my_service();
};

```

The next step after the mapping is to generate the skeleton and the stub of the components to be implemented by Java implementations.

`MyDemo.java` is a bootstrap of the application to initialize the ORB, obtain the Name Service, obtain component servers `ComponentServer`, obtain a container home, instantiate a container on each server, install homes for `Client` and `Server`, create a components with `create()` method of homes, configure all components, and connect each client to the server by calling the `configuration_complete()` methods of the components implementation.

One of the goals of CCM is to make programming easier by the composition of components and the extension of component. The partial of `myDemo.java` code below illustrates the composition of CCM components, where `c1`, `c2`, and `c3` are instances of the client component and `s` is an instance of the server component. The receptacle ports of `Client` component instances `c1`, `c2`, `c3`, connect to the facet port of `Server` component instance `s`. The `c1`, `c2`, `c3` also use their sink port to consume the event published by `Server` component instance `s` through its event source port. The `provide_my_service()` and `subscribe_to_clients()` are mapped from `provide MyService my_service` and `publishes TextEvent to_clients` in `Server` component interface, and the `connect_my_service()` and `get_consumer_from_servers()` are mapped from `uses Myservice my_service` and `consumes TextEvent from_servers` in the component `Client` respectively.

```

...
org.omg.Components.CCMHome h = null;
h = server1_cont.install_home("myDemo",
"org.objectweb.ccm.myDemo.monolithic.ClientHomeImpl.create_home",
config);
ClientHome ch = ClientHomeHelper.narrow(h);
...
h = server2_cont.install_home("myDemo",
"org.objectweb.ccm.myDemo.monolithic.ServerHomeImpl.create_home",
config);
ServerHome sh = ServerHomeHelper.narrow(h);
...

Server s = sh.create();
Client c1 = ch.create();
Client c2 = ch.create();
Client c3 = ch.create();

c1.name("saleman");
c2.name("bidder1");
c3.name("bidder2");
s.name("The Server");

// Connecting clients and consumers to server.

MyService my_service = s.provide_my_service();
MyService my_service = s.provide_my_service();
c1.connect_my_service(my_service);
c2.connect_my_service(my_service);
c3.connect_my_service(my_service);

```

```
s.subscribe_to_clients(c1.get_consumer_from_servers());
s.subscribe_to_clients(c2.get_consumer_from_servers());
s.subscribe_to_clients(c3.get_consumer_from_servers());
...
...
```

`push_to_clients()` pushes a `TextEvent` to all registered event consumers(listeners) and this method is implemented by `Client`, and `push_from_event()` specifies the response to an incoming notification of `TextEvent` and this method is implemented by `Server`. Here is a simple implementation of `push_from_servers()` in package `org.objectweb.ccm.myDemo.monolithic` of `ClientImpl.java`.

```
...
public void push_from_servers(TextEvent event)
{
    textArea_.append(event.text + "\n");
}
...
...
```

The server implementation in package `org.objectweb.myDemo` of `ServerImpl.java` implements the `show` method for the `MyService` interface by showing a string text and pushing events to the client sink by `push_to_clients(...)`, which is defined in `ServerCCM.java`. An internal timer event is triggered out when time is out, which results in sending a `TextEvent` to notify the client.

```
...
public void show(String name, String stringNumber)
{
    int intNumber = Integer.parseInt( stringNumber);

    textArea_.append(name + ":" + intNumber + "\n");

    if(flag == true)
    {
        salePrice = intNumber;
        timer.start();
        flag = false;
    }
    else
        result = getBiggestNumber(intNumber, name);
}

...
timer = new javax.swing.Timer(FIVE_SECOND, new
    java.awt.event.ActionListener(){
    public void actionPerformed(java.awt.event.ActionEvent
        evt)
        {
            if(result < salePrice)
            {
                get_context().push_to_clients(new TextEventImpl("Bid too
                    low")));
            }
        }
    });
...
}
```

```

        textArea_.append("\n" + "Time is out" );
        textArea_.append("\n" + "Bid is too low" + "\n");}
    else
    {
get_context().push_to_clients( new TextEventImpl("Sold at $ "
" + result) );
        textArea_.append("\n" + "Time is out" );
        textArea_.append("\n" + "Sold at $ " + result +
"\n");
    }
}
});
timer.setRepeats(false);

```

In `ClientImpl.java`, when a client pushes the “send” button, an `actionEvent` will be triggered and `actionPerformed()` is invoked. It will get a reference to a Server component by `get_connection_my_service()` method and invoke the method `show()` of that component.

```

...
public void actionPerformed(ActionEvent evt)
{MyService service = the_context_.get_connection_my_service();
...
service.show(name, inputField.getText());
...

```

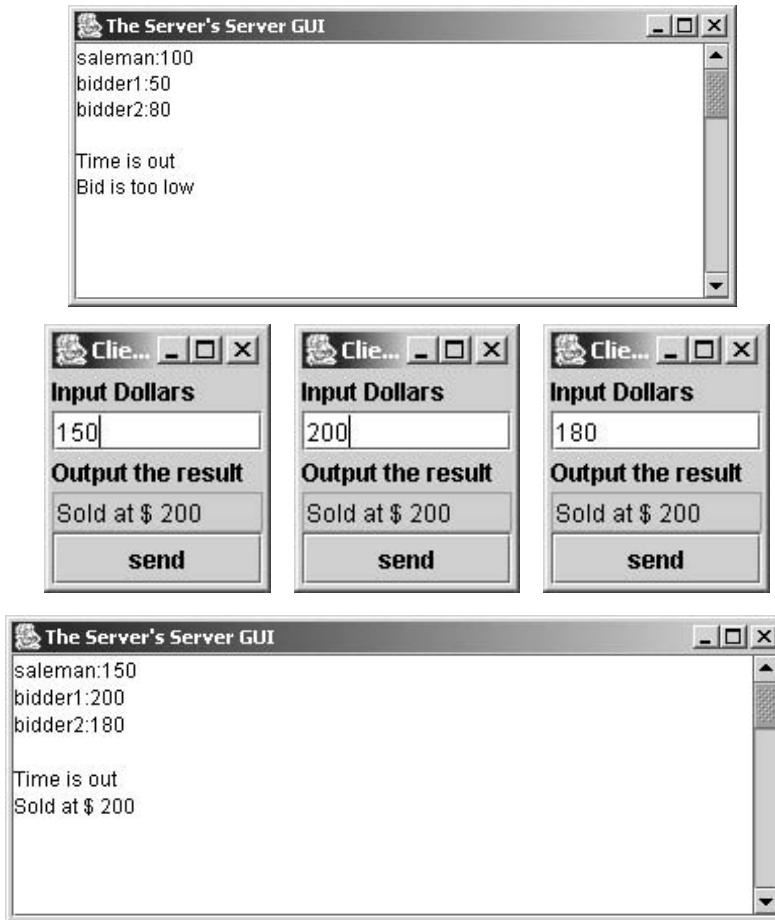
The class in `ClientHomeImpl.java` inherits from the local `CCM_ClientHome` interface. It implements the `create_home()` method called by the OpenCCM Component Server to create a home instance.

The class in `ServerHomeImpl.java` implements the `create_home()` method to create a home instance and register the `TextEvent` valueType factory to the ORB.

After Java CORBA 2 stubs and Java OpenCCM skeletons are generated, we can compile all Java implementation sources, build archive `myDemo.jar`, start installation, start up `nameService`, start up server component, and run the application. The details are shown in the section of “Practice Lab.”

A sample output of this application is shown below. The starting price is set at \$100. The next two bids are too low. In order to simplify the discussion, we assume that the first client is a seller and other clients are bidders.





The highest in next three bids is \$200. After time is over, 200 is posted to all clients.

5.3 CONNECTION MODEL OF CORBA AND CCM

CCM components are reusable building blocks used to build a new component. Developers can glue multiple components together to build a new application or a new component. In this section, we discuss the connection and the communication between CORBA components.

There are two types of ports: import and export.

The import ports are receptacles, which require or get services from other components, and sinks, which receive incoming events from event sources in other components.

The export ports are facet ports, which provide interface services to other components, and event publisher/emitter ports, which send out events when the trigger condition is met.

The facets and receptacles work together in a pair in synchronous communication mode. Publisher/emitter and sink work in pair in asynchronous mode. We have

seen an example in Figure 5.5, where the connections between **Client** and **Server** components are established by two channels. One is a synchronous connection from receptacle port of *Client* component to facet port of *Server* component and the other is an asynchronous connection from event source port of *Server* component to the sink port of *Client* component.

The synchronous method invocation expects a result back from the method invocation right away. The asynchronous communication does not expect to get response immediately; instead, it can continue its work without being idle in waiting until being notified by the server. There are many ways to implement such a callback method, which passes in a callback method as an argument of a remote method invocation to a server component and lets the server component call back this callback method when it is ready. Other options can be implemented just as the mode in the pair of event source and event sink in CCM.

Compositions of components versus programming is a goal of CCM, which can accelerate time-in-market and reduces the cost. Figure 5.6 shows the connection between four components by all four types of component ports.

```
interface itemService{
    string name();
    setStarting_price(in float sell);
    setCurrent_price(in float bid);
    getCurrent_price();}

eventtype statusEvent{
    public string status; }

eventtype timeEvent{
    public long timeLeft; }

interface RegistrationService{
    register(String name);
```

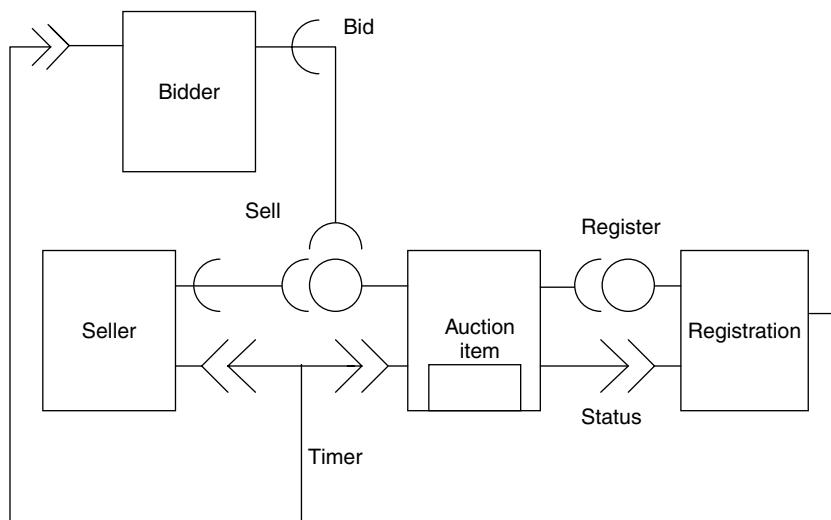


FIGURE 5.6. Connections between CCM components.

```

        unRegisterset(String name);}
component Auction_item supports itemService{
    provides itemService myItemService;
    uses registrationService myItemregistration
    consumes TimerEvent from_Registration;
    publishes statusEvent itemInfoI;
    emits statusEvent itemInfoII;
    attribute String name;
    attribute float start_price;
}
valuetype NameKey:
::Components::PrimaryKeyBase{
public string name;};

home Auction_itemHome manages Auction_item primarykey NameKey{
factory create_Auction_item(in string name);
finder find_Auction_item(in string name);};

```

The *statusInfo* and *TimerEvent* are event types. *RegistrationService* is an interface, which is shown later.

```

component Registration supports RegistrationService{
    provides RegistrationService myRegistration;
    consumes statusEvent itemInfoR;
    publishes TimerEvent timeInfoR;
}
home RegistrationHome manages Registration{...};

component Bidder{
    uses itemService myItemService
    consumes TimerEvent timeInfoB;
    consumes StatusEvent itemInfoB
}
home bidderHome manages Bidder{...};

component Seller{
    uses itemService myItemService
    consumes StatusEvent itemInfoS;
}
home SellerHome manages Seller{...};

```

The following incomplete fragment shows the connection between *bidder*, *Auction_item*, and *registration* implementation. Assume *item1* is an instance of component type *Auction_item* created by *create()* method of home manager. The *bidder1* is an instance of the *Bidder* component type and *registration1* is an instance of the component type *Registration*.

```

//provide_myItemService() is mapped from provides ItemService
//myItemService() in Auction_item

```

```

ItemService item=item1.provide_myItemService();

//connect_myItemService() is mapped from uses ItemServicein
//Bidder component
bidder1.connect_myItemService(item);

//subscribe_itemInfoI is mapped from publishes statusEvent
//itemInfoI in Auction_item
//get_consumer_itemInfoR is mapped from consumes statusEvent
//itemInfoR in Registration

item1.subscribe_itemInfoI(registration1.get_consumer_iteminfoR());

```

The connection configuration can be programmed in a bootstrap module, which is just like the OpenCCM sample shown in last section. It can also be configured in an assembly descriptor, which will be discussed in next section.

5.4 DEPLOYMENT MODEL OF CORBA AND CCM

One of the most important features of software component is its reusability and portability that a component can be deployed on any compliant server. It is platform and programming language independent.

The first step in the CCM component deployment is the component packaging. Before getting a component in a package, we must get the component implementation, which is shown in Figure 5.7. A component package is stored in an archive file with a packaging tool. A packaged component is self-descriptive by an XML descriptor.

Next step may be the assembling. A component package can be assembled with other related component packages together by a CORBA assembly tool.

Finally, a component assembly can be deployed with a deployment tool and run at any compliant server [OMG CCM 2002; Flissi 2002].

Here is a sample component descriptor files for Client component and Server component that we discussed before.

```

<?xml version ="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "...corbacomponent.dtd">
<corbacomponent>
  <corbaversion> 3.0 </corbaversion>
  <componentrepid repid="IDL:myDemo/Client" />
  <homerepid repid="IDL:myDemo/Client" />
  <componentkind>
    <process>
      <servant lifetime="container" />
    </process>
  </componentkind>
  ...
  <configurationcomplete set="true" />
  ...
  <homefeatures name="ClientHome" repid="IDL:myDemo/Client">
  </homefeatures>

```

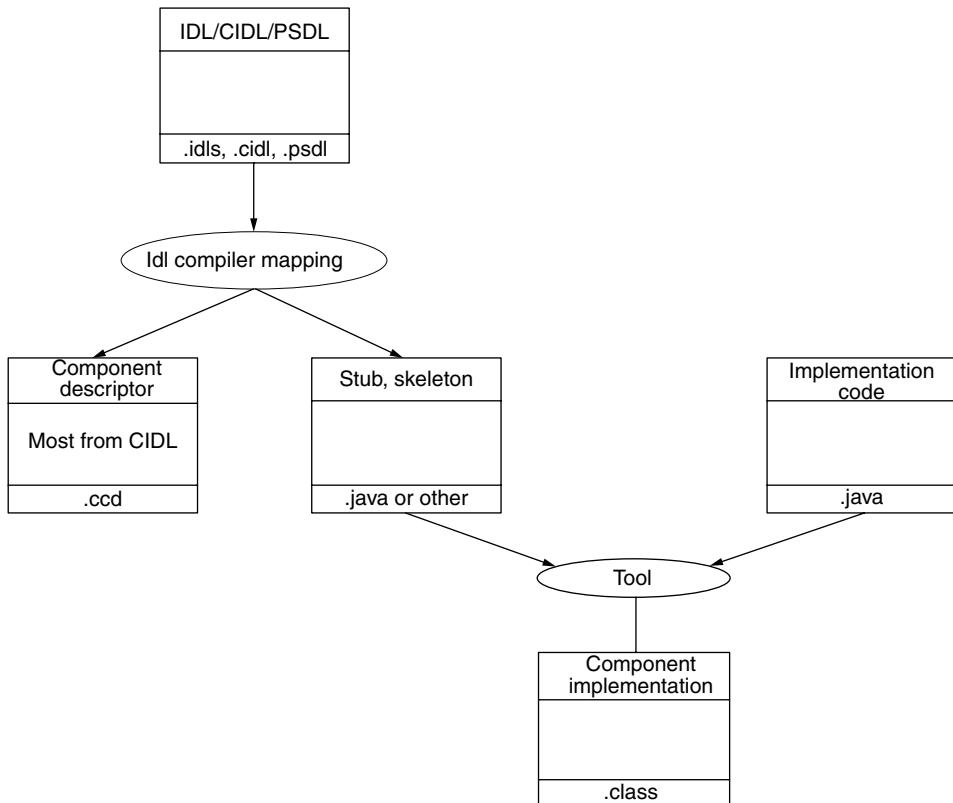


FIGURE 5.7. Component implementation of IDL, CIDL, PSDL.

```

<componentfeatures name="Client" repid="IDL:myDemo/Client">
  <ports>
    <uses usesname="my_service" repid="IDL:myDemo/Server" />
    <consumes consumesname="from_servers" eventtype="TextType">
      <eventpolicy policy="normal" />
    </consumes>
  </ports>
</componentfeatures>
...
</corbacomponent>
  
```

Here is a sample ccd file for the Server component in myDemo example.

```

<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "....corbacomponent.dtd">
<corbacomponent>
  <corbaversion> 3.0 </corbaversion>
  <componentrepid repid="IDL:myDemo/Server" />
  <homerepid repid="IDL:myDemo/Server" />
  
```

```

<componentkind>
  <session>
    <servant lifetime="component" />
  </session>
</componentkind>
...
<homefeatures name="ServerHome" repid="IDL:myDemo/ServerHome">
</homefeatures>
<componentfeatures name="ForkManager" repid="IDL:myDemo/Server">
  <ports>
    <provides providesname="my_service"
      repid="IDL:myDemo/Server"
      facettag="1">
    </provides>
    <publishes publishesname="to_clients" eventtype="TextType"
      <eventpolicy policy="normal"/>
    </publishes>
  </ports>
</componentfeatures>
...
</corbacomponent>

```

5.4.1 Packaging

A CCM component must be packaged in a zip package such as a. jar file that is different from classic CORBA object, which does not require packaging. A component package has one component consisting of one or many implementations for different OS or programming languages; one IDL file of the component; one CORBA Component Descriptor (.ccd), which has the information generated from CIDL for container management; one Property File Descriptor (.cpf), defining pairs of name/default value for CCM component's attributes and home properties; and one Software Package Descriptor (.csd), describing the package general elements (title, author, description, Web page, license, link to IDL file, etc.) and a list of implementations (information about implementations such as OS, ORB, language, compiler, dependencies on other libraries, entry point, etc.).

Figure 5.8 shows the CCM package generation.

The following is a sample Component Software package Descriptor CSD file for the Server component. The IDL definition and implementation of this Server component are specified in it. Also, all related CCDs in this software package are listed here.

```

<?xml version="1.0"?>
<!DOCTYPE softpkg SYSTEM ".../src/dtd/ccm/softpkg.dtd">

<softpkg name="Server" version="2,0">
  <title>Server</title>
  <pkgtype>CORBA Component</pkgtype>
  <author>
    ...
  </author>
  <description>A simple client/server application</description>

```

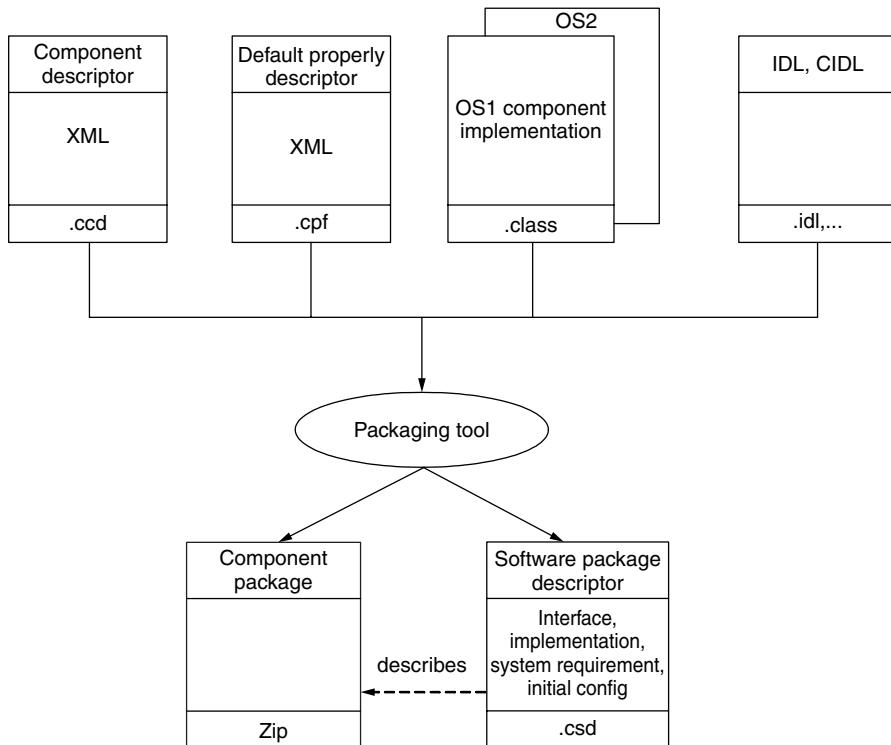


FIGURE 5.8. CCM component package.

```

<license href="http://corbaweb.lifl.fr/OpenCCM/COPYRIGHT" />
<idl id="IDL:myDemo/Server:1.0">
  <link href="myDemo.idl3"></link>
</idl>
<descriptor type="CORBA Component">
  <fileinarchive name="META-INF/server.ccd" />
</descriptor>

<implementation id="ServerImpl">
  <os name="Win2000" version="X.X.X.X" />
  <os name="Linux" version="X.X.X.X" />
  <processor name="x86" />
  <compiler name="JDK" />
  <programminglanguage name="Java" />
  <dependency type="ORB" action="assert">
    <name>OpenORB</name>
  </dependency>
  <code type="Java class">
    <fileinarchive name="archives/myDemo.jar" />
  <entrypoint>org.objectweb.ccm.myDemo.cif.
    ServerHomeImpl.create_home</entrypoint>
</code>
  
```

```

<runtime name="Java VM" version="1,4.0.0"/>

</implementation>
</softpkg>
```

Similarly, the Client software package descriptor is listed as follows.

```

<?xml version="1.0"?>
<!DOCTYPE softpkg SYSTEM ".../src/dtd/ccm/softpkg.dtd">

<softpkg name="Client" version="2,0">
  <title>Client</title>
  <pkgtype>CORBA Component</pkgtype>
  <author>
    ...
  </author>
  <description>A simple client/server application</description>
  <license href="http://corbaobjectweb.lifl.fr/OpenCCM/COPYRIGHT"/>
  <idl id="IDL:myDemo/Client">
    <link href="myDemo.idl3"></link>
  </idl>
  <descriptor type="CORBA Component">
    <fileinarchive name="META-INF/Client.ccd"/>
  </descriptor>

  <implementation id="ClientImpl">
    <os name="Win2000" version="X.X.X.X"/>
    <os name="Linux" version="X.X.X.X"/>
    <processor name="x86"/>
    <compiler name="JDK"/>
    <programminglanguage name="Java"/>
    <dependency type="ORB" action="assert">
      <name>OpenORB</name>
    </dependency>
    <code type="Java class">
      <fileinarchive name="archives/myDemo.jar"/>
    </code>
  </implementation>
</softpkg>
```

5.4.2 Component Package Assembly

The CCM component assembly is a zip archive file (such as a jar file) that is for deployment of a group of connected components. Each component package assembly is described by a CORBA Component Assembly Descriptor (.cad), which describes all contained Component Software package Descriptor (.csd). A component assembly package contains one or many component packages, property file descriptors (.cpf)

specifying initial attribute values, and a Component Assembly Descriptor (CAD) specifying home instance, component instance, and connections between ports in connection of multiple components. Each assembly may have one or more component packages. The `cad` file references all related `.csd` and specifies how a home and a component should be instantiated. An assembly may be imported to be reused or extended. Figure 5.9 shows the generation of a CCM component assembly. A component package assembly can then be deployed at a CORBA server.

The following is an example CAD XML file from our `mydemo` OpenCCM application discussed before.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE componentassembly SYSTEM "... my.dtd">
<componentassembly id="myDemo">
  <componentfiles>
    <componentfile id="Client">
      <fileinarchive name="META-INF/client.csd">
        </fileinarchive>
    </componentfile>
    <componentfile id="Server">
      <fileinarchive name="META-INF/server.csd">
        </fileinarchive>
    </componentfile>
  </componentfiles>
  <partitioning>
    <homeplacement cardinality="1" id="ServerHome">
      <componentfileref idref="Server"/>
    </homeplacement>
  </partitioning>
</componentassembly>
```

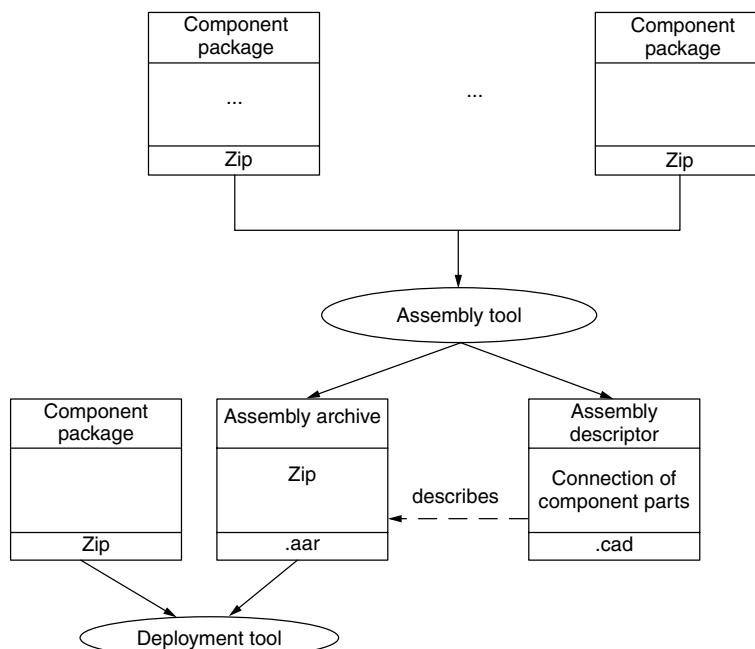


FIGURE 5.9. CCM component assembly.

```

<componentimplref idref="ServerImpl"/>
<registerwithhomefinder name="OpenCCM/ServerHome" />
<registerwithnaming name="OpenCCM/ServerHome" />
<componentinstantiation id="Server">
    <componentproperties>
        <fileinarchive name="META-INF/server.cpf">
            </fileinarchive>
    </componentproperties>
</componentinstantiation>
<destination>ComponentServer2</destination>
</homeplacement>
<hostcollocation>
    <homeplacement cardinality="1" id ="ClientHome">
        <componentfileref idref="Client"/>
        <componentimplref idref="ClientImpl"/>
        <registerwithhomefinder name="OpenCCM/ClientHome" />
        <registerwithnaming name="OpenCCM/ClientHome" />
        <componentinstantiation id="C1">
            ...
        </componentinstantiation>
        <componentinstantiation id="C2">
            ...
        </componentinstantiation>
        <componentinstantiation id="C3">
            ...
        </componentinstantiation>
    </homeplacement>
    <destination>ComponentServer1</destination>
</hostcollocation>
</partitioning>
<connections>
    <connectinterface>
        <usesport>
            <usesidentifier>my_service</usesidentifier>
            <componentinstantiationref idref="C1"/>
        </usesport>
        <providesport>
            <providesidentifier>my_service</providesidentifier>
            <componentinstantiationref idref="Server"/>
        </providesport>
    </connectinterface>
    <connectinterface>
        <usesport>
            <usesidentifier>my_service</usesidentifier>
            <componentinstantiationref idref="C2"/>
        </usesport>
        <providesport>
            <providesidentifier>my_service</providesidentifier>
            <componentinstantiationref idref="Server"/>
        </providesport>
    </connectinterface>
    <connectinterface>
        <usesport>

```

```

<usesidentifier>my_service</usesidentifier>
<componentinstantiationref idref="C3" />
</usesport>
<providesport>
    <providesidentifier>my_service</providesidentifier>
    <componentinstantiationref idref="Server" />
</providesport>
</connectinterface>
<connectevent>
    <consumesport>
        <consumesidentifier>from_servers</consumesidentifier>
        <componentinstantiationref idref="C1" />
    </consumesport>
    <publishesport>
        <publishesidentifier>to_clients</publishesidentifier>
        <componentinstantiationref idref="Server" />
    </publishesport>
</connectevent>
<connectevent>
    <consumesport>
        <consumesidentifier>from_servers</consumesidentifier>
        <componentinstantiationref idref="C2" />
    </consumesport>
    <publishesport>
        <publishesidentifier>to_clients</publishesidentifier>
        <componentinstantiationref idref="Server" />
    </publishesport>
</connectevent>
<connectevent>
    <consumesport>
        <consumesidentifier>from_servers</consumesidentifier>
        <componentinstantiationref idref="C3" />
    </consumesport>
    <publishesport>
        <publishesidentifier>to_clients</publishesidentifier>
        <componentinstantiationref idref="Server" />
    </publishesport>
</connectevent>
</connections>
</componentassembly>
```

We can see that all CSD file names and all connections of all component instances are specified in this CAD file.

5.4.3 CCM Deployments

Next to the assembling is the component deployment, which is a one-step automatic deployment. An assembly archive is deployed by a deployment tool. Figure 5.10 shows the process of CCM component deployment, including installation of component server, container, home on host, and CCM component object instantiation. Any CCM component object starts its life cycle after being instantiated.

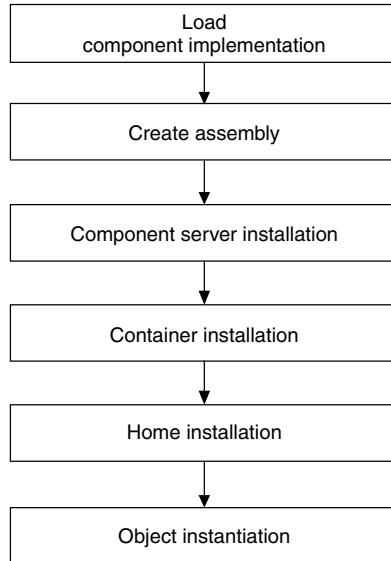


FIGURE 5.10. CCM component deployment.

5.5 EXAMPLES AND LAB PRACTICE

5.5.1 Lab 1: Classic CORBA Component of Temperature Converter

Step 1: Compilation of IDL Interface in SDK1.4.x

1. Change to the directory that contains the file Convert.idl shown in 5.2.1.
2. Run the IDL-to-Java Compiler

> **idlj – fall Convert.idl**

After compilation, there are the following source codes that will be inside the subfolder TempConvertApp, just as the snapshot shown below.

```

C:\ Command Prompt
Volume in drive F is hongwei
Volume Serial Number is FCB8-3071
Directory of F:\summer2003\kai\cobra
[.]           [...]          Convert.idl
ConvertClient.java   ConvertServer.java   [TempConvertApp]
      3 File(s)           5,146 bytes
      3 Dir(s)   3,948,507,136 bytes free
F:\summer2003\kai\cobra>cd t*
F:\summer2003\kai\cobra\TempConvertApp>dir/w
Volume in drive F is hongwei
Volume Serial Number is FCB8-3071
Directory of F:\summer2003\kai\cobra\TempConvertApp
[.]           [...]          Convert.java
ConvertHelper.java   ConvertHolder.java   ConvertOperations.java
ConvertPOA.java      _ConvertStub.java
      6 File(s)           8,457 bytes
      2 Dir(s)   3,948,507,136 bytes free
F:\summer2003\kai\cobra\TempConvertApp>
  
```

3. Compile all the .java files using the following command

➤ **javac *.java TempConvertApp*.java**

Step 2: Start orbdd daemon

➤ **start orbdd – ORBInitialPort 1050**

Step 3: Start the Server From an MS-DOS system prompt, enter

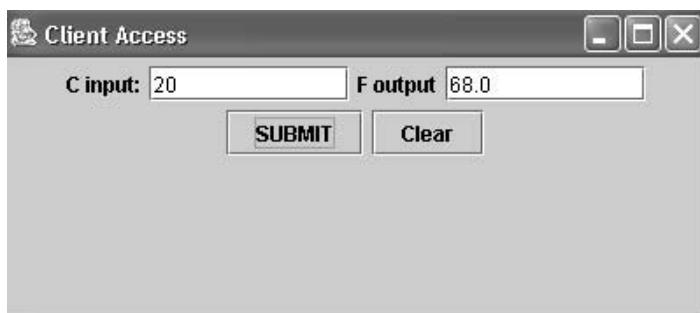
➤ **start java ConvertServer – ORBInitialPort 1050 – ORBInitialHost localhost**



Step 4: Run the Client Application Open one more MS-DOS prompt and enter

➤ **java ConvertClient – ORBInitialPort 1050 – ORBInitialHost localhost**

The GUI will show up; randomly type in one number into the left textfield Celsius and click the “submit” button and the converted Fahrenheit result will be shown in the right one, as the following.



5.5.2 Lab 2: CCM CORBA Component of MyDemo in OpenCCM

MyDemo is discussed in detail in this chapter. We explore all details of OpenCCM installation; first, environment configuration and next, CCM component compilation, packaging, and deployment. Finally, we will show the execution of this CCM application. We use OpenCCM 0.6 version, which is the newest version at the time of writing this book [Flissi 2002; OpenCCM 2003].

Step 1: Installations of all Requirements

1.1 A full Java product CORBA 2.4 ORB

Download OpenORB-1.3.0.zip from http://prdownloads.sourceforge.net/openorb/OpenORB-1.3.0.zip?use_mirror=unc.

Extract the package to the drive you prefer, for example, D:\OpenORB-1.3.0.
Put “orb.properties” in the right directory, for example,

```
D:\ > cd OpenORB-1.3.0\lib  
D:\OpenORB-1.3.0\lib> java -jar openorb-1.3.0.jar
```

```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

H:>>d:
D:>>cd Openo*
D:>cd OpenORB-1.3.0>cd lib
D:>OpenORB-1.3.0\lib>java -jar openorb-1.3.0.jar
File "D:\j2sdk1.4.1_02\jre\lib\orb.properties" exists.

D:>OpenORB-1.3.0\lib>
```

1.2 Install Java Development Kit j2sdk1.4.

1.3 Use ant 1.5-beta2 tool to build OpenCCM platform.

Go to <http://jakarta.apache.org/ant/>. Download the latest version Ant 1.5.1. and install “ant” in D:\apache-ant-1.5.3-1 and set environment variables:

```
set ANT_HOME d:\apache-ant-1.5.3-1
set path d:\apache-asnt-1.5.3-1\bin
```

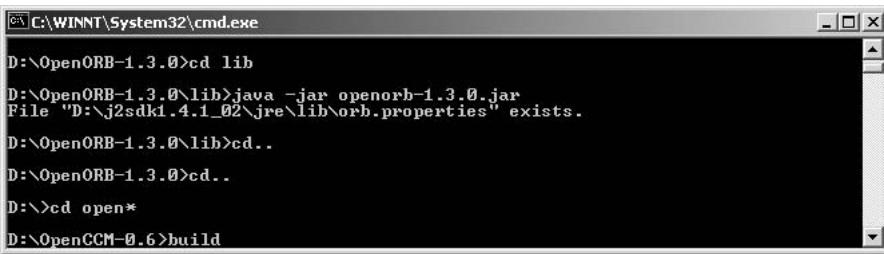
1.4 Download the OpenCCM package.

Go to <http://www.objectweb.org/openccm/download/index.html>, download and extract the OpenCCM package in D:\OpenCCM-0.6.

Step 2: Configuration

2.1 Create a build.properties template file.

```
D:\OpenCCM-0.6> build
```



```
C:\WINNT\System32\cmd.exe
D:\OpenORB-1.3.0>cd lib
D:\OpenORB-1.3.0\lib>java -jar openorb-1.3.0.jar
File "D:\j2sdk1.4.1_02\jre\lib\orb.properties" exists.
D:\OpenORB-1.3.0\lib>cd..
D:\OpenORB-1.3.0>cd..
D:\>cd open*
D:\OpenCCM-0.6>build
```

2.2 Edit `build.properties` file to configure following environment variables:

```
# For OpenORB-1.3.0 for Java.
ORB.name=OpenORB-1.3.0

# The directory where the OpenCCM Platform will be installed.
# By default, this is in the distribution directory.
OpenCCM.root.dir=D:/OpenCCM-0.6
install.dir=D:/OpenCCM-0.6/build

# The directory where the used ORB is installed.
ORB.home.dir=D:/OpenORB-1.3.0

# The directory where the Transaction Service binaries are installed.
OTS.home.dir=D:/OpenCCM-0.6

# The directories where the used Transaction Service jar files are installed.
OTS.jar.dir=D:/OpenCCM-0.6/externals/ots
```

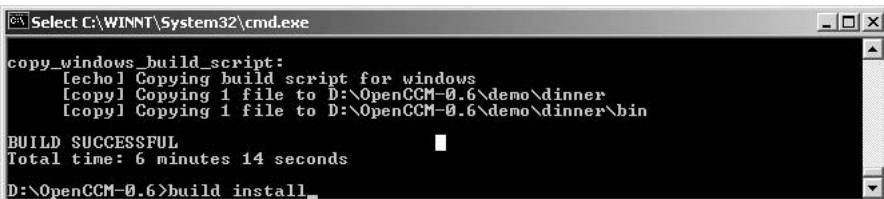
Step 3: Compilation and Installation

3.1 Build OpenCCM platform based on the configured `build.property` file.
D:\OpenCCM-6.0>build



```
C:\WINNT\System32\cmd.exe
check_unix_os:
copy_unix_build_script:
check_windows_os:
copy_windows_build_script:
[echo] Copying build script for windows
BUILD SUCCESSFUL
Total time: 1 minute 27 seconds
D:\OpenCCM-0.6>build
```

3.2 **D:\OpenCCM-6.0>build install**



```
Select C:\WINNT\System32\cmd.exe
copy_windows_build_script:
[echo] Copying build script for windows
[copy] Copying 1 file to D:\OpenCCM-0.6\demo\dinner
[copy] Copying 1 file to D:\OpenCCM-0.6\demo\dinner\bin

BUILD SUCCESSFUL
Total time: 6 minutes 14 seconds
D:\OpenCCM-0.6>build install
```

3.3 Loading the OpenCCM environment.

D:\OpenCCM-6.0\build>cd bin

D:\OpenCCM-6.0\build\bin>call envi_OpenCCM.bat

```
[copy] Copying 1 file to D:\OpenCCM-0.6\build\xml
BUILD SUCCESSFUL
Total time: 23 seconds
D:\OpenCCM-0.6>cd build\bin
D:\OpenCCM-0.6\build\bin>call envi_OpenCCM.bat
D:\OpenCCM-0.6\build\bin>
```

Step 4: Run the demo

4.1 D:\OpenCCM-6.0\demo\demo3>bin\start_java

```
D:\OpenCCM-0.6\build\bin>cd..
D:\OpenCCM-0.6\build>cd..
D:\OpenCCM-0.6>cd demo
D:\OpenCCM-0.6\demo>cd demo3
D:\OpenCCM-0.6\demo\demo3>bin\start_java
```

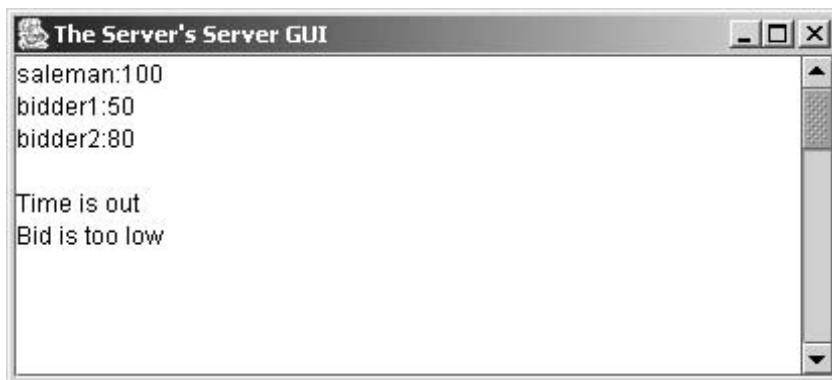
4.2 Repeat step 1 again.

D:\OpenCCM-6.0\demo\demo3>bin\start_java

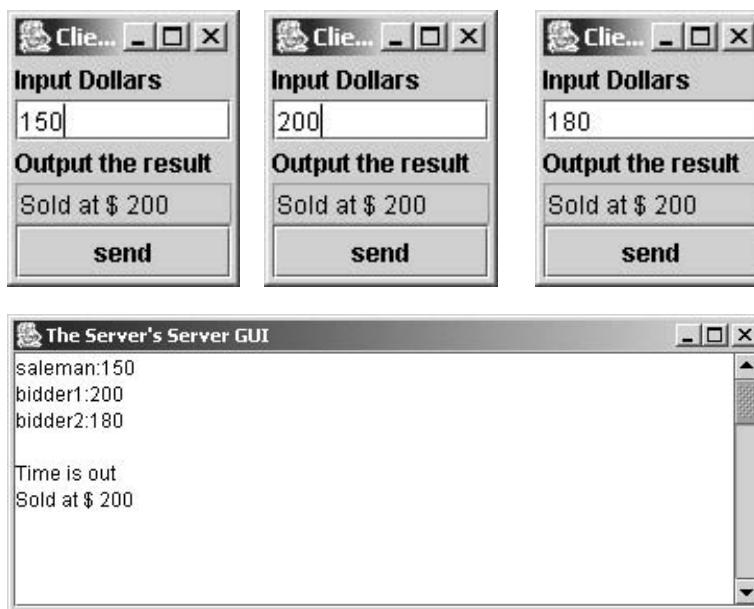
```
build_archive:
[echo] Building demo\demo3\archives\demo3.jar
[jar] Building jar: D:\OpenCCM-0.6\demo\demo3\archives\demo3.jar
all:
BUILD SUCCESSFUL
Total time: 1 minute 40 seconds
D:\OpenCCM-0.6\demo\demo3>bin\start_java
```



In this demo, the first client sets the selling price at \$100. Two bidders' bids are too low.



Two more bids are \$150, \$200, and \$180 in sequence. After time is out, the final sold price is \$200 since it is the highest bid.



5.6 SUMMARY

This chapter introduces the CORBA component-based distributed computing. CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use

to work together over networks. Using the standard protocol IIOP and ORB, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

CCM is the first open standard in true component-based software engineering for distributed computing. This chapter discusses how to design, implement, package, deploy, execute, and reuse CORBA component. The difference of CORBA 3.x from CORBA 2.x is discussed. “CORBA 2” sometimes refers to CORBA object model and “CORBA 3” refers to the CCM. The CCM model makes the distributed software development more productive, more reusable, and more easier to compose new components or applications. A CCM component must be packaged, and can be assembled and deployed as a reusable component.

The *separation of interface from implementation*, enabled by OMG IDL, is the essence of CORBA. An IDL interface is a contract between client and implementation. A CCM component interface exposes all its ports. It has its facets, receptacles, event sources, event sinks, and configurable attributes. It is very easy to connect multiple CCM components by their ports.

CCM model has shown its strength in component-based software development.

5.7 SELF-REVIEW QUESTIONS

1. Which following file extensions are used in CCM component packaging, assembling, and deployment?
 - a. .csd
 - b. .cad
 - c. .cpf
 - d. None of above
 - e. All
2. CORBA 2.x standard is a true component model for distributed computing.
 - a. True
 - b. False
3. A CORBA-distributed component and its client may be run
 - a. in different machines
 - b. in same machine
 - c. both
 - d. on HTTP
4. Both of CORBA 2.x and CORBA 3.x support ports of sink, source, facet, receptacle.
 - a. True
 - b. False
5. Attribute is available in
 - a. CORBA 2.x only

- b. CORBA 3.x only
 - c. both of a and b
- 6. CCM CORBA home interface provides
 - a. component lifetime management
 - b. business logic processing
 - c. both
 - d. neither
- 7. Communications between CCM CORBA components can be
 - a. synchronous
 - b. asynchronous
 - c. both
- 8. A CORBA client extends a CORBA skeleton to marshal and unmarshal the arguments of a remote method invocation of a distributed CORBA component and result returned from this invocation.
 - a. True
 - b. False
- 9. CORBA 3.x is integrated with EJB 2.x.
 - a. True
 - b. False
- 10. CORBA client uses a naming service to locate the target object by its deployment name.
 - a. True
 - b. False

Keys to Self-Review Questions

1. e 2. b 3. c 4. b 5. c 6. a 7. c 8. b 9. a 10.a

5.8 EXERCISES

1. What is CORBA?
2. What is ORB?
3. Who produces CORBA?
4. Why use CORBA?
5. What is OpenCCM?
6. What is CORBA object reference?
7. Is CORBA object reference persistent?
8. What is IIOP?
9. What is DII?

- 10.** What is DSI?
- 11.** What is IR?
- 12.** What is SII
- 13.** What is SSI?
- 14.** What is OA?
- 15.** What is the difference between BOA and POA?
- 16.** What is synchronous delegate?
- 17.** What does a stub do?
- 18.** What does a skeleton do?
- 19.** What can Facet port and Receptacle port do in CCM?
- 20.** What can an Attribute in CCM do?
- 21.** What is the detail process of CCM packaging, assembling, and deployment?

5.9 PROGRAMMING EXERCISES

- 1.** Design a Java CORBA component that provides services of a calculator. The calculator has the functionality to perform addition, subtraction, multiplication, and division of two real numbers. It can also detect a zero divisor in division operation.
- 2.** Design a GUI client of this calculator in Java.
- 3.** Design a MicroCCM or OpenCCM CORBA component that provides services of a calculator. The calculator has the functionality to perform addition, subtraction, multiplication, and division of two real numbers. It can also detect the zero divisor in division operation.
- 4.** Design a GUI client of this calculator in Java.
- 5.** Implement the CORBA temperature conversion example in an interoperable mode, that is, Component and its client use different programming languages, one in C++ and the other in Java.
- 6.** Extend the bidder CCM example program to an operational auction system. The system takes the offer from the seller and sets the item name, description, starting price, bid duration, amount, and contact information. The system also takes the bids from customers with their contact information, bid price, and bid time. After time out, the bidder will be notified with the result and all bidders will be notified of the status of the auction processing.

REFERENCES

[OMG CCM 2002] OMG CCM Implementers Groups, *CORBA Component Model Tutorial*, CCM/02-04-01, www.omg.org, 2002.

- [Flissi 2002] Flissi, A. *Getting Started with OpenCCM Tutorial*, 2002.
- [OpenCCM 2003] *OpenCCM User Guide, Installation Guide*, <http://openccm.objectweb.org>, 2003.
- [Component 2004] <http://www.componentsource.com>, 2004.

6

.NET COMPONENTS

Objectives of This Chapter

- Introduce .NET framework
- Introduce the concepts of .NET components
- Discuss the types of .NET components, connections of components, and their deployments
- Distinguish local and distributed components
- Distinguish aggregation and containment compositions
- Distinguish synchronous and asynchronous method invocations
- Provide step-by-step tutorials on building, deploying, and using .NET components

6.1 .NET FRAMEWORK

6.1.1 Overview of .NET Framework

The .NET framework is one of the newest technologies introduced by Microsoft Corporation. Its first Beta version was released in 2000. The .NET framework is a platform for rapid and easier building, deploying, and running secured .NET software components to be integrated in applications as well as for rapid developing XML Web Services and applications. It provides a highly productive, component-based, multilanguage environment for integrating existing applications with the Internet to meet the challenges of new applications for deployment and operation of internet-scale applications. The .NET framework encompasses a virtual machine that provides a new

platform for software development. The core of the .NET framework includes XML and Simple Object Access Protocol (SOAP) to provide Web Services over the Internet.

The purpose of the .NET framework is to facilitate the developments of desktop window, and Web-based application services on Windows platform and make them available and accessible not only on Windows platform but also on other platforms through common protocols such as SOAP and HTTP.

First, .NET simplifies the componentization especially for Component Object Model (COM), Distributed COM (DCOM) technology. Although COM components can be reused as plug-and-play software components in component software and application constructions, the development process is too complex and COM does not support versioning (side-by-side execution), which may cause version conflict (DLL Hell problem). The .NET technology supports component assembly deployment that allows multiple versions of same-named components to coexist without any conflict. .NET technology simplifies the creation and deployment of components in addition to securing reliable and scalable services provided by components.

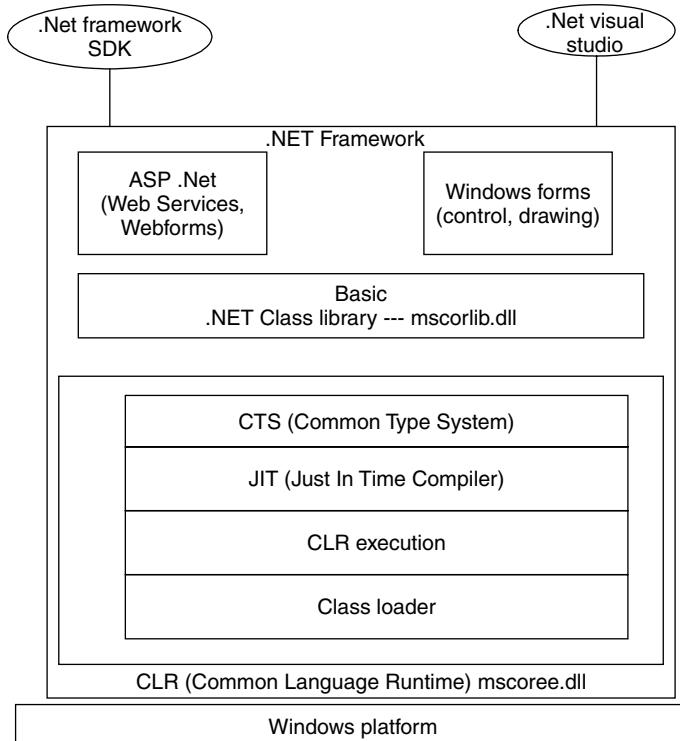
Second, .NET facilitates developments of distributed components by Remoting Channel technology. .NET framework supports the interoperability between COM and .NET components. The XML Web Service is another kind of component. A .NET component can work with any existing COM components. In other words, .NET can provide services to COM components, and COM components can also be used by any .NET components. It is much easier to develop components in .NET than in COM. Web service is a replacement of MS DCOM technology for Internet applications supported by XML, SOAP, and HTTP protocols. .NET frees developer's coding from heavy enterprise programming such as transaction management through Enterprise Service. .NET Web Service overcomes DCOM's lack of support for firewall and makes services available across platforms via loosely coupled XML and SOAP protocols.

The .NET framework is available in .NET Framework SDK and Visual Studio .NET IDE SDK, both of which can be downloaded from MS Website. The .NET Framework SDK is the foundation of Visual Studio .NET and is a part of Visual Studio .NET when Visual Studio .NET is installed. The .NET framework consists of two main parts (as shown in Figure 6.1): Common Language Runtime (CLR) and a set of unified framework basic class libraries including ASP.NET Web forms for building Web applications, Windows Forms for building desktop applications, and ADO.NET for data access. The SDK includes all your needs to write, build, test, and deploy .NET applications. It supports all .NET languages such as VB .NET, VC .NET, C#, and others. .NET SDK and Visual Studio .NET can access services of all layers in the .NET framework platform.

6.1.2 Foundation of .NET Framework – CLR

CLR is a virtual machine environment sitting on the top of Windows operating system. CLR consists of Common Type System (CTS), Just-In-Time IL Compiler (JIT), Execution unit, plus some other management services such as garbage collection and security management. CLR is like the JVM in Java. All these software components are assembled in a package of assembly (just like Java archive file .jar file) that consists of MS Intermediate Language (MSIL) code and manifest (Metadata about this packet).

The IL code is translated into native code by JIT compiler in CLR. IL code is verified by CTS first to check the validity of data type used in the code. Figure 6.2 shows how the CLR works.

**FIGURE 6.1.** .NET framework.

.NET framework integrates multiple programming languages (VB, managed VC++, C#, etc.) by CLR implementation. Not only a component in one language can access the services provided by other components in other languages but also a class in one language can inherit properties and methods from related classes in other Languages. The United Class Library provides a set of reusable classes for component development. The CTS defines a standard set of data types and rules for creating new types. The CLR knows how to execute these types. There are two categories of types: Reference type and Value type. The code targeting CLR and to be executed by CLR is called *.NET managed code*. All MS language compilers generate managed codes that conform to the CTS.

The IL code is like Java byte code. Regardless of the types of source code programming languages, IL code can communicate with each other through the support of CLR. The IL code can be either in the format of executable (.EXE) or Dynamic Link Library (.DLL). If these IL code are generated by .NET compiler, they are called *managed code*. The managed code can be executed only on .NET aware platform. Some DLL or EXE generated by non .NET compilers (such as early version of VC++) are called *unmanaged code*.

In summary, the CLR is a high-performance execution engine. It provides a code-execution environment that manages the code targeting the .NET framework. The code management includes management of memory, thread, security, code verification, and IL compilation [Platt 2003; Chappel 2002].

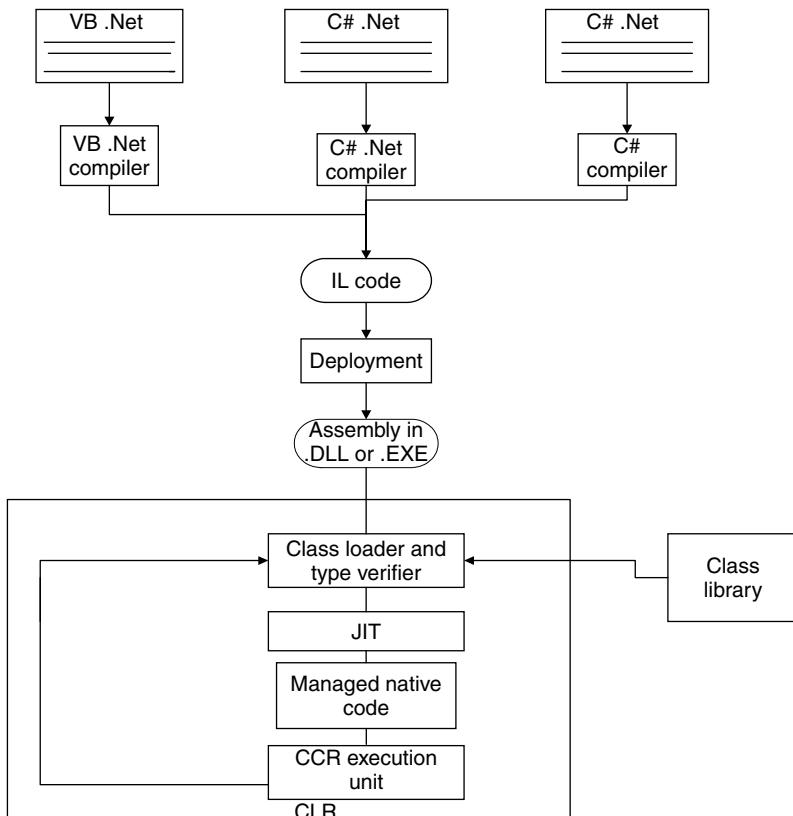


FIGURE 6.2. .NET CLR.

6.1.3 .NET Framework Class Library

The .NET framework class library is a collection of reusable basic classes that are well organized by namespaces. The framework class library collects all classes including Windows Foundation Classes (WFC) into a unified set of classes, which is a single set of classes. The namespace is just like a package in Java technology and the class library is just like the Java API structure. A namespace consists of many classes and subnamespaces. It is deployed as a component class library itself and is organized in a component-based hierarchy. Figure 6.3 lists a partial set of components in the .NET class library hierarchy. All these classes in the library can be used by other classes in different languages.

The root namespace in the class library is System namespace, which contains many basic classes such as Object, Console, and may contain subnamespaces such as IO, Net, Data, Remoting, etc. For example, XML is a subnamespace of System namespace that is deployed as System.XML.dll, ADO.NET is available in System.Data.dll that corresponds to System.Data namespace, and Form-based UI classes are available in System.Windows.Forms.dll corresponding to namespace System.Windows.Forms.

Developers can create custom namespace and organize related classes in a custom namespace. A namespace can be deployed as an assembly of binary components.

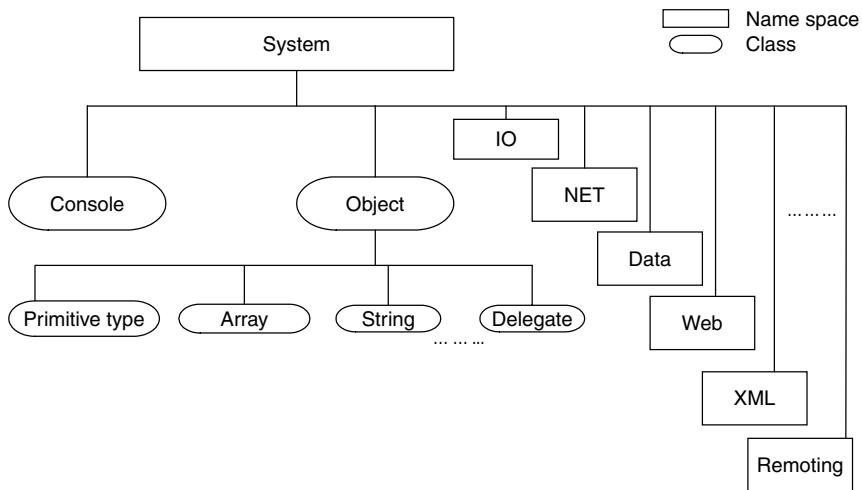


FIGURE 6.3. Portion of .NET class library.

Classes with the same name can be placed in different namespaces because they are referenced by different namespace prefix.

In order to use classes in a namespace, a directive `using <namespace>` in C# or `import <namespace>` in VB must be included at the beginning of code. The system built-in basic class library is deployed in `mscorlib.dll` file [Platt 2003; Chappel 2002].

6.2 COMPONENT MODEL OF .NET

The .NET Component technology has enhanced and simplified existing MS COM, DCOM, COM+ technologies. MSIL DLL components are replacing COM Components, the MSIL Remoting Channels EXE components are replacing DCOM Component, and the Web Service components are new SOAP components intended to be cross-platform and cross-language Web-based components. .NET components are much easier to develop than COM and DCOM. They resolve the COM's version conflict DLL Hell problem and firewall problem in DCOM.

Also, the .NET component technology is unified language oriented. Any .NET component is in the format of precompiled MSIL, which can be binary plugged-in by any other MSIL components or any other .NET compatible clients.

The .NET framework itself is built up in a component model, for example, System namespace `System.Runtime.Remoting` is available in `mscorlib.dll` and `System.XML` namespace is available in `System.XML.dll`. A .dll file is a .NET deployed component (Assembly). A namespace is just like a logical package in Java to organize related classes together. An assembly may have many namespaces, and one namespace may span over multiple assembly file. The details of .NET assembly will be discussed in the following sections.

A .NET component is a single precompiled and self-described MSIL module built from one or more classes or multiple modules deployed in a DLL assembly file. An assembly consists of up to four parts:

1. Manifest (table of info records): name of assembly, key info version, strong name, culture info, files that make up assembly, reference depended assemblies exported info.
2. Metadata of modules.
3. IL code of modules.
4. Resources such as image files.

A module has MSIL code and its metadata but without manifest. A module is not loadable dynamically. It is used as a building block at the compile time to build up an assembly Module file. It has an extension of **.netmodule**. There may be one or many classes in a module. An assembly is made up by one or many classes in a module. Each module may be coded in different languages but finally in same MSIL format. Assembly has a manifest file to self-describe the component itself. An assembly has a file extension **.dll** or **.exe** and is dynamically loadable. That is why most people say .NET component is an assembly (we will say it is deployed in an assembly). A **.dll** file is not executable, just like a class file is a byte code file that is not executable.

There are many different types of components in the .NET framework. We can classify them into visual or nonvisual component categories. A visual component is a control that can be deployed in a toolbox as an icon for “drag and drop” in a window form container. We focus on nonvisual component, which is known as *.NET component*.

A .NET component can be installed at client site, server site, or middleware site. It does not matter what kind of component it is; a .NET component always provides some services to its clients (client may be another component or client application).

Figure 6.4 shows the contents of an assembly.

A .NET component can be a local component (**.dll**), which can only be accessed locally (within the same application domain), in the same machine or a remote (distributed) component (**.exe**), which can be accessed remotely (across application domains) in same machine or different machines. An application domain is a lightweight process, which can be started or stopped independently within a process. It is just another level of the isolation in .NET. The idea is very similar to out-of-process DLL prior to .NET. One component cannot directly access a component in another application domain or process because each application domain has its own memory space.

A .NET DLL component can be deployed as a private component, which knows the target client or can be deployed as a shared public component, which does not know the target client. A DLL component can be plugged-in to Windows form, Web form of another DLL or application [MSDN 2004; Component 2004; Lowy 2003].

Let us take a look at a very simple sample component in C#. This component provides services of converting temperature between F° and C°.

//Listing of TempConvComp.cs:

```
using System;
namespace TempConv
{ public class TempConvComp
    { public double cToF (double c)
        { return (int) ((c*9)/5.0+32);
        }
    public double fToC (double f)
```

```
        { return (int) ((f-32)*5/9.0);  
    }  
}
```

We can build a module from it with the following command:

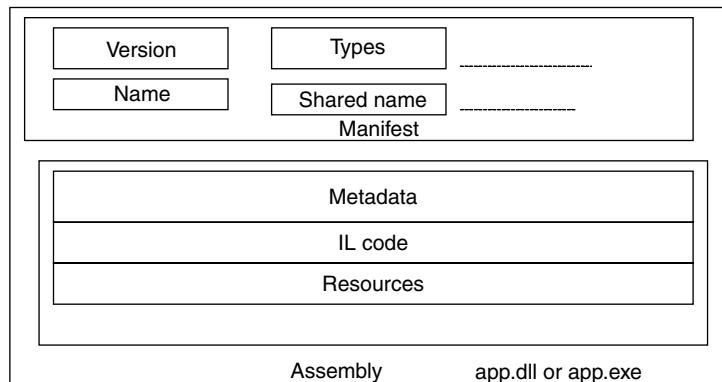
```
>csc /t:module TempConvComp.cs → TempConvComp.netmodule
```

This module can be added to a component by

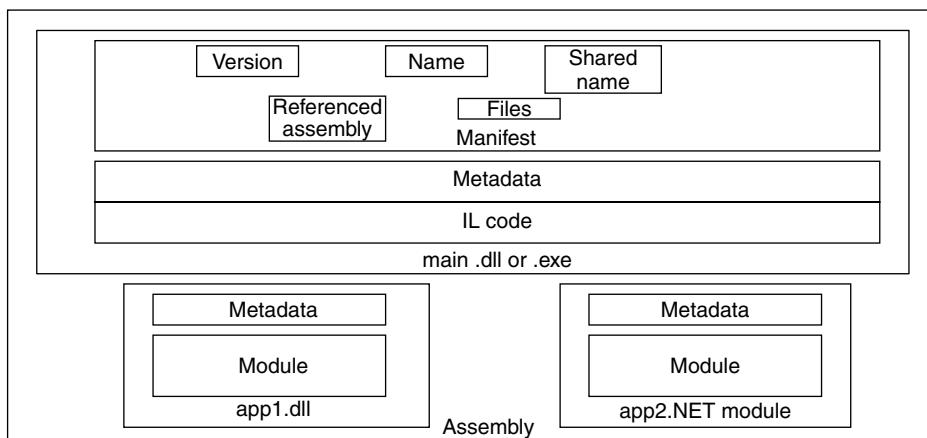
```
>csc /t:library /addmodule: TempConvComp.netmodule anotherComp.dll
```

Also, we can build a DLL component by the command

```
>csc /t:library TempConvComp.cs→TempConvComp.dll
```



(a) Single file assembly



(b) Multiple file assembly

FIGURE 6.4. Single file assembly and multiple file assembly for .NET components.

Here are two clients of this component. One is `TempConvCSCClient.cs` in C# and the other is `TempConvCppClient.cpp` in C++. Both of them reuse this `TempConvComp` component.

The following list is the C# program list of the client of `TempConvComp` component:

```
using System;
using TempConv;

class MainApp
{
    public static void Main()
    {
        TempConv.TempConvComp myCSTempConvComp =
        new TempConv.TempConvComp();
        double choice;
        double input;
        double output;
        bool next = true;

        while (next)
        {
            Console.WriteLine("Please enter your choice:
                                1 - Converter from F to C,
                                2 - from C to F,
                                3 - exit");
            choice=Double.Parse(Console.ReadLine());
            if (choice == 1)
            {
                Console.WriteLine("Please tell me the temperature in F:
                                ");
                input=Double.Parse(Console.ReadLine());
                output = myCSTempConvComp.fToC(input);
                Console.WriteLine(output);
            }
            else if (choice ==2)
            {
                Console.WriteLine("Please tell me the temperature in C:
                                ");
                input=Double.Parse (Console.ReadLine());
                output = myCSTempConvComp.cToF(input);
                Console.WriteLine(output);
            }
            else
            {
                next= false;
                Console.WriteLine ("See you next time.");
            }
        }
    }
}
```

In `TempConvCSClient.cs`, the client loads `TempConv` namespace by “using `TempConv`” directive and instantiates an instance of the `TempConvComp` component and invokes `fToC()` and `cToF()` methods provided by this component.

The Client application in C# can be built by the following command:

```
>esc /t:exe /r:TempConvTemp.dll TempConvCSClient.cs →
TempConvCSClient.exe
```

The following is the managed C++ program list of the client of `TempConvComp` component

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"
#include <iostream>

#using <mscorlib.dll>
#using "TempConv.dll"

using namespace System;
using namespace std;

// This is the entry point for this application
#ifndef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
    int choice;
    double input;
    double output;
    bool next = true;
    TempConv::TempConvComp * myCSConvComp =
        new TempConv::TempConvComp ();

    while (next) {
        Console::WriteLine(S"Please enter your choice:
                           1 - Converter from F to C,
                           2 - from C to F,
                           3 - exit");
        cin>>choice;
        if (choice == 1) {
            Console::WriteLine(S"Please input the temperature in
                           F: ");
            cin>>input;
            output = myCSConvComp->fToC(input);
            Console::WriteLine(output);

        }
        else if (choice == 2){
```

```

Console::WriteLine(S"Please input the temperature in
C: ");
    cin>>input;
    output = myCSConvComp->cToF(input);
    Console::WriteLine(output);
}
else {
    next= false;
    Console::WriteLine (S"See you next time.");
}
}
return 0;
}

```

Similarly, in `TempConvCppClient.cpp`, the client loads namespace `TempConv` by `#using "TempConv.dll"` and gets an instance of this component, and then gets the services provided by this component.

In summary, the client of a component makes a reference to the server component at compiler time, and then the client loads the component dynamically into its application domain at run time when it is needed. Figure 6.5 depicts the processing of a component at compiling time and at run time [MSDN 2004; Lowy 2003].

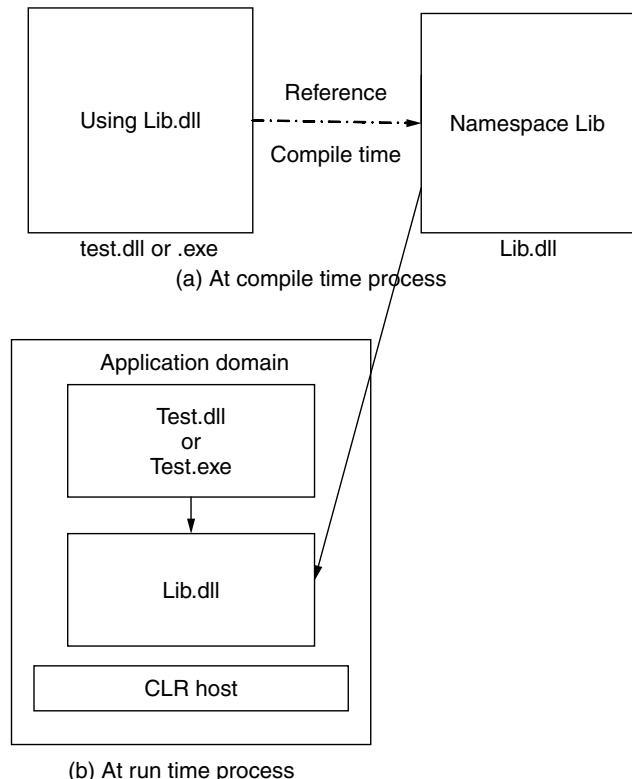


FIGURE 6.5. .NET components at compiling time and at run time.

6.3 CONNECTION MODEL OF .NET

6.3.1 .NET Component Compositions

Component compositions enable the component reuse in either aggregation compositions or containment compositions.

In aggregation composition model, the service of inner component hands out its service directly to the client of outer component. In aggregation composition, the outer component exposes the interfaces of the inner component. The `innerM()` method of inner component becomes part of interface to the outer component as shown in Figure 6.6. The detailed implementation example is shown in the following code fragment.

In containment compositions, if a request to the outer component needs help from an inner component, the request is forwarded to that inner component. The outer component does not expose the interface of the inner component. The containment is transparent to the client of an outer component. The client is blind to the handler of the request. The `outerM2()` delegates a request to the `innerM()` method of inner component as shown in Figure 6.7.

A .NET component can also be composed by mixed aggregations and containments in a flat structure or nested compositions in multiple levels in depth.

Here, we use an example to explain the concepts of combined containment and aggregation compositions.

```
using System;
namespace NS1
{
    public class Inner
    {
        public void innerM ()
        {
            Console.WriteLine ("I am Inner.");
        }
    }
}

using System;
using NS1;

public class Outer
{
    public Inner i = new Inner ();
    //aggregation: Outer expose the Inner
    public void outerM1 ()
    {
        Console.WriteLine ("I am outer.");
    }
    public void outerM2()           //delegation in containment
    {
        i.innerM();
    }
    public static void main()
```

```

{
    outer o1 = new Outer();
    Inner i1 = o1.i;
    i1.innerM(); //interface to the aggregate
    o1.outerM();
    o1.outerM2(); // interface to the containment
    Inner i2 = new Inner();
    i2.innerM();
}
}

```

Figure 6.8 shows a direct method invocation of a component.

6.3.2 Communication of Components by Event and Delegate

The .NET Delegate is a method type (a reference to a method) that is similar to function pointer in C++, but it is type-safe and secure. A Delegate will delegate a flow control to its registered event handler when the event is raised. It works in the pattern of observer, which is a kind of an event listener in Java.

An instance of a Delegate can hold a static method of a class, or a method of a component, or a method of object itself.

There are two types of Delegates: **SingleCast** and **MultiCast**.

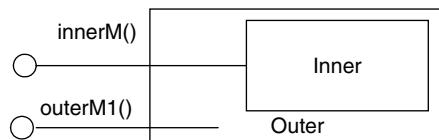


FIGURE 6.6. Aggregation.

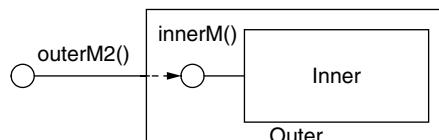


FIGURE 6.7. Containment.

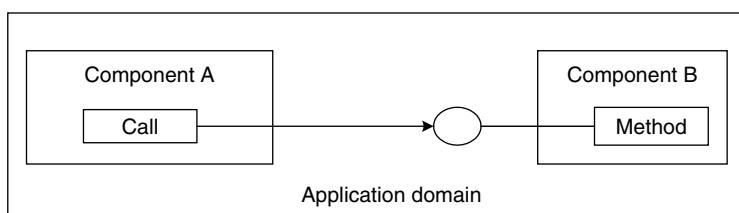


FIGURE 6.8. Direct invocation.

A SingleCast Delegate can only delegate one method at a time seen in the following example:

```
Delegate int Mydelegate();
public class MyClass
{ public int ObjMethod() { - - - }
static public int StaticMethod () { - - - }
public class Drive { Static public void Main()
{ Myclass c = new MyClass();
MyDelegate dlg = new MyDelegate(c.ObjMethod());
dlg();
dlg = new MyDelegate (MyClass.StaticMethod());
dlg();
}
}
```

As seen in this example, `MyDelegate` is a Delegate that references any method with `int` return type and without any parameter. The signatures of `ObjMethod` and `StaticMethod` match the Delegate `MyDelegate`. The first `dlg()` invokes `ObjMethod` and the second `dlg()` invokes class method `StaticMethod`.

A MultiCast Delegate has a void return type and can bind multiple methods, and it will invoke them in the order of registrations.

```
Delegate void MultiDelegate();
MultiDelegate mdlg = null;
mdlg += new MultiDelegate (Method1);
mdlg += new MultiDelegate (Method2);
```

Registration is done by `+=` Delegate operation and unregistration is done by `-=` operation.

The Delegate plays a role of listener, and the event handler must register this listener to be able to handle the event once the event is fired off. The relationship between the event handler and event trigger via a delegate is shown in Figure 6.9.

An event is a message sent by an object to invoke an action. The object that raises the event is event source and the object that intercepts the event and handles the event is event target.

This is an event-driven communication model between components or within the same component. The Delegate class is the communication channel class between the event source and event target. Event can be a predefined event such as an event trigger by a Windows Form component. A developer can also define a custom event. The procedure to create and use a delegate event is listed here.

1. Create a delegate class.

```
Public delegate void DelegateStart();
```

2. Create a class containing a delegate field, Class `MyClass`.

```
{ public event DelegateStart EventStart;
```

```
- - -
```

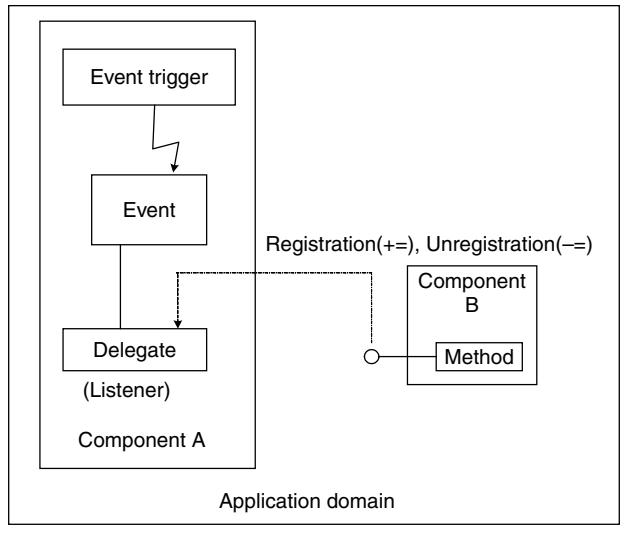


FIGURE 6.9. Delegate wiring of a delegate event and its handler.

3. Define an event handler.

```
public void handleEvent(){ - - -}
```

4. Bind delegate event with an event handler via event listener, trigger an event, invoke the event handler.

```
Public static void Main() { MyClass EventObj = new
    MyClass();
    EventObj.EventStart += new DelegateStart(handleEvent);
    EventObj.EventStart();
    ...
}
```

6.3.3 Remoting Connectors for .NET Distributed Components

A component or a client cannot directly access a remote component running in a different application domain in the same or different processes unless it uses Remoting channel connection. The marshaling makes it possible to invoke a remote method of a distributed component. There are two ways to marshal an object: in MBV (Marshal by Value), the server passes a copy of an object to client or in MBR (Marshal by Reference) the client creates a proxy of a remote object. When a remote component must run at a remote site, MBR is the only choice.

A channel is the way to implement communications between clients and remote components. We use TCP channel in our example. The channel is similar to the socket communication in Java. Each channel must bind with a port. A client channel binds a client port, and the server channel binds a port on the server. We create TCP channel

on port 4000 and register the channel with the remote class and URI name, which will be used by client to get an object of remote component. We also need to create a TCP channel and register it on the client site.

Here is an example of a distributed component and its client. They are running in a remote mode (different application domains or different processes). We use the same component TempConvComp but we build it as a distributed component to be accessed remotely at this time.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class CoTempConv : MarshalByRefObject
{
    public static void Main()
    {
        TcpChannel channel = new TcpChannel(4000);
        ChannelServices.RegisterChannel(channel);

        RemotingConfiguration.
            RegisterWellKnownServiceType (
                typeof(CoTempConv),
                "TempConvCompDotNet",
                WellKnownObjectMode.Singleton);
        System.Console.WriteLine("Hit <enter> to
            exit...");
        System.Console.ReadLine();
    }

    public double cToF(double c)
    {
        return (int)((c*9/5.0+32)*100)/100.0;
    }

    public double fToC(double f)
    {
        return (int)((f-32)*5/9.0*100)/100.0;
    }
}
```

Here is the client of the above component.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

class MainApp
{
    public static void Main()
    {
```

```
try
{
    TcpChannel channel = new TcpChannel();
    ChannelServices.RegisterChannel(channel);
    CoTempConv myCSTempConvComp =
        ((CoTempConv)Activator.GetObject(
            typeof(CoTempConv),
            "tcp://127.0.0.1:4000/
                TempConvCompDotNet"));

    double choice;
    double input;
    double output;
    bool next = true;

    while (next)
    {
        Console.WriteLine("Please enter your
            choice:
            1 - Converter from F to C,
            2 - from C to F,
            3 - exit");
        choice=Double.Parse (Console.ReadLine());
        if (choice == 1)
        {
            Console.WriteLine("Input temperature in F:
                ");
            input=Double.Parse(Console.ReadLine());
            output = myCSTempConvComp.fToC(input);
            Console.WriteLine(output);
        }
        else if (choice ==2)
        {
            Console.WriteLine("Input temperature in C:
                ");
            input=Double.Parse(Console.ReadLine());
            output = myCSTempConvComp.cToF(input);
            Console.WriteLine(output);
        }
        else
        {
            next= false;
            Console.WriteLine ("See you next time.");
        }
    }
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

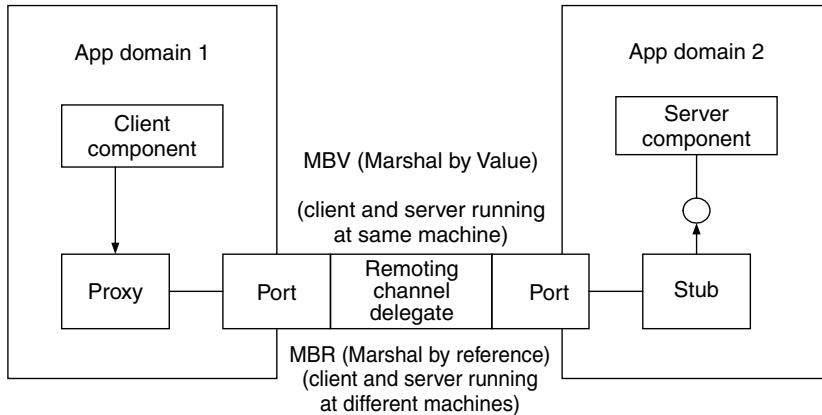


FIGURE 6.10. Remote synchronous method invocation of a distributed component.

The following shows the steps to build the server and client.

```
>csc TempConvComp.cs
>csc /r:TempConvComp.exe TempConvCSClient.cs
```

Now it is ready to activate the distributed server component and its client by running the following commands:

```
>TempconvComp.exe
>TempConvCSClient.exe
```

The client gets a reference to remote component by `Activator.GetObject()` and invoke the method of this remote component. The diagram of Figure 6.10 shows the remote synchronous method invocation of a distributed component.

6.3.4 Remoting Asynchronous Callback Invocation between Distributed .NET Components

The Remoting asynchronous callback is based on Remoting Delegate. It will not block out the client while waiting for notification from remote components. For example, suppose someone wants to be notified once the stock prices reach a specified level. Instead of polling the stock price all the time, why not let the server notify you and you can do whatever you want to do. In some other cases, the jobs on the server will take very long to complete, why not let the server notify you when the job is done.

Let us reuse the `TempConvComp` component discussed before and make asynchronous callback from server to client. It looks like a round trip. When the client makes a synchronous call to remote method of remote component, it passes a callback method to server to be called back late through Remoting Delegate.

Here are two Delegates: one is `MyDelegate` pointing to the remote method “`cToF`” of remote component named “`TempConvDotNET`.” The other asynchronous Delegate is `AsynchCallback`, which is passed to `BeginInvoke` method of `MyDelegate`.

```

using System;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class Client {

    public delegate double MyDelegate(double c)
    public static int main(string [] agrs)

    TcpChannel chan = new TcpChannel();
    ChannelServices.RegisterChannel(chan);
    CoTempConv obj =
        (CoTempConv)Activator.GetObject(typeof(CoTempConv),
        "tcp://localhost:4000/TempConvCompDotNet");
    If(obj == null) System.Console.WriteLine("Could not locate
        server");
    else {
        AsyncCallback cb = new AsyncCallback(Client.MyCallBack);
        MyDelegate d = new MyDelegate(obj.cToF);
        IAsyncResult ar = d.BeginInvoke(32, cb, null);
    }

    System.Console.WriteLine("Hit <enter> to exit ... ");
    System.console.ReadLine();
    return 0;
}

public static void MyCallBack(IAsyncResult ar)
{
    MyDelegate d = (MyDelegate)((AsyncResult)ar).AsyncDelegate;
    Coinsole.WriteLine(d.EndInvoke(ar));
    Coinsole.WriteLine("See you next time");
}
}

```

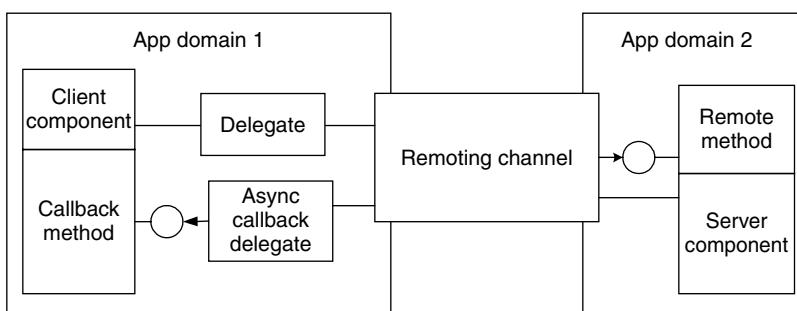


FIGURE 6.11. Asynchronous callback method invocation of distributed component.

We can find two Delegates here. One is `MyDelegate` pointing to the remote method “`cToF`” of distributed component; the other is `AsyncCallback` pointing callback method “`MyCallBack`.” Figure 6.11 illustrates the asynchronous callback method invocation of distributed component discussed above.

The first parameter of `BeginInvoke` is a 32 degree in Celsius for “`cToF`” remote method and the second parameter is a callback Delegate.

The callback will not block client program. When the distributed component completes the conversion work, the callback method is called and `IAsyncResult` is returned back to client [Csharp 2004; Component 2004; Code 2004].

6.4 .NET COMPONENT DEPLOYMENTS

A .NET component can be deployed as private component or public shared component in an assembly file. The assembly is an atomic deployment (distribution) unit in .NET Framework. A private component knows the component where it will be plugged-in. A public shared component does not know which component will use itself. It must be published (registered) in a centralized repository Global Assembly Cache (GAC). A shared component supports side-by-side multiple version component execution.

6.4.1 Private Deployment

A private component must be deployed in a same directory of its client or subdirectory of where the client is. It is the simplest deployment done by copying all components to where the client is. The undeployment is done by simply removing all related .dll components from the directory. A private component does not support versioning control and just for in-house development within a company. Figure 6.12 shows an example of private deployment.

```
>csc /t:library /out:dir1\comp1.dll comp1.cs
>csc /t:library /out:dir2\comp2.dll comp2.cs
>csc /r:dir1\comp1.dll, dir2\comp2.dll /out:client.exe client.cs
```

An XML configuration description file is required if components are not located in the same directory with their client. The private path must be specified in the probing

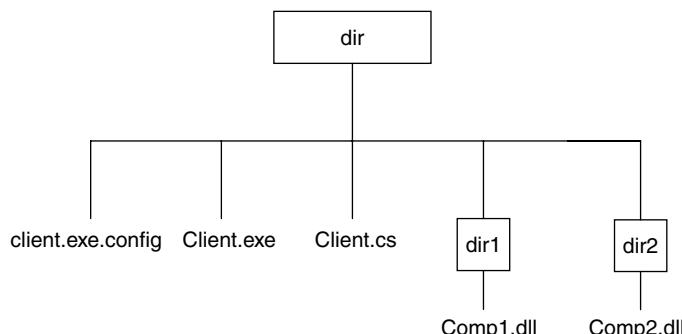


FIGURE 6.12. Directory structure of a private component deployment.

subtag of assembly building tag in the application configuration file with an extension of “config.”

Here is a sample of configuration file:

```
<configuration>
  <runtime>
    <assemblyBinding>
      <probing privatePath = "dir1; dir2"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

The CLR will use a private path to look for the required component to load if the CLR cannot find the required component in current directory.

6.4.2 Public Shared Deployment

The most popular reusable component deployment is to deploy a component with a strong name, to register it with GAC. A shared component with strong name can make itself unique by public/private key pair. A shared component registered with GAC can make itself to be shared anywhere.

The steps needed to create a shared .NET component are as follows:

Step 1: Create a pair of public key/private key by **sn.exe** utility.

```
>sn -k mykey.snk
```

The public key is for verification against private key, which is signed as a cryptographic signature stored in component assembly. The public key is stored in a manifest of the assembly. When a client references the component, the public key token is stored in the client’s assembly.

Step 2: Embed the following lines into the source code of component:

```
using System.Reflection;
[assembly:AssemblyKeyFile ("mykey.snk")]
[assembly:AssemblyDelaySign (false)]
[assembly:AssemblyVersion ("1,2,3,4")]
```

The following command will sign the signature with the component immediately without a delay:

```
>csc /t:library mycomponents.cs
```

The next command line will store a public key token in the client component.

```
>csc /r:mycomponent.dll /out:myapplication.exe myapplication.cs
```

If the signature delay is needed, we can sign the signature late by

```
>sn -R mycomponent.dll mykey.sn
```

The signature is verified when the component is registered with GAC in step #3 to ensure that the component is not altered since the assembly was built.

At run time, the public key token in the client manifest is verified against the public key that is part of component identity. If they match, then it indicates that this is the right component wanted.

Figure 6.13 shows the private and public key pair in manifests of the component and its client component.

The version number of a shared component is marked by four discrete numbers separated by dots in the format of “**major.minor.build.revision**.” A side-by-side execution is implemented by .NET versioning. The .NET framework allows different versioned components with the same name running side-by-side in different applications. The CLR checks the major and minor numbers first. By default, only an exact match on **major.minor** is allowed. The build number is backward compatible by default. In other words, Version “1.2.3.0” is compatible with “1.2.0.0” but “1.2.3.0” is not compatible with “1.2.4.0.” If the version number in a component’s manifest cannot match any component in GAC, it will load a versioned component that is different in revision part. The revision number is called Quick Fix Engineering (QFE), which is always compatible by default.

The default version policy can be customized by overriding the assembly specification in configuration file. For example,

```
<assemblyIdentity>
    <binding Redirect oldVersion = "1.2.3.4"
              newVersion = "2.0.0.0"/>
</assemblyIdentity>
```

This setting indicates the component in version “2.0.0.0” will be loaded instead of “1.2.3.4.”

Step 3: Register the shared component in GAC.

>gacutil /i mycomponent.dll

Step 4: Using shared component.

The client must make a reference to the shared component to be reused.

>csc /t:exe /r:mycomponent.dll /out:myapp.exe myapp.cs

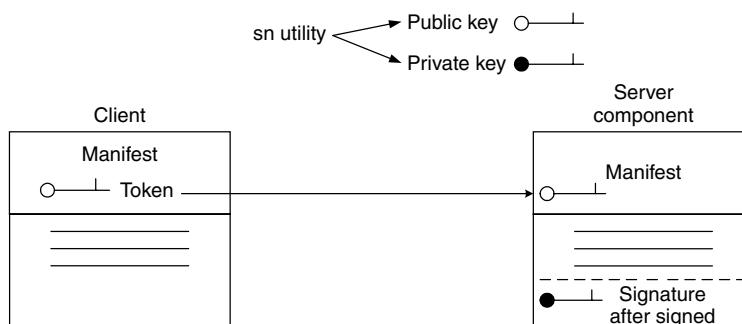


FIGURE 6.13. Public/private key pair in .NET component.

In order to reuse the shared component, the client source code must “use namespaces” where namespace is available in the assembly. An example of shared component deployment of the TempConvComp component is shown in the following:

```
using System;
using System.Reflection;
[assembly:AssemblyVersion("1.0.0.0")]
[assembly:AssemblyKeyFile("originator.key")]

namespace TempConv
{

    public class TempConvComp
    {
        public TempConvComp()
        {
        }

        public double cToF(double c)
        {
            // how to control output format
            return (int)((c*9/5.0+32)*100)/100.0;
        }

        public double fToC(double f)
        {
            // how to control output format
            return (int)((f-32)*5/9.0*100)/100.0;
        }
    }
}
```

The following steps show how to build a shared component:

```
>sn -k originator.key
>csc /t:library /out:TempConv.dll TempConvComp.cs
>gacutil /i TempConv.dll
>csc /r:TempConv.dll /t:exe /out:TempConvCSClient.exe
    TempConvCSClient.cs
```

The code of the client program is the same as `TempConvCSClient.cs` in Section 6.2.

6.5 VISUAL STUDIO .NET

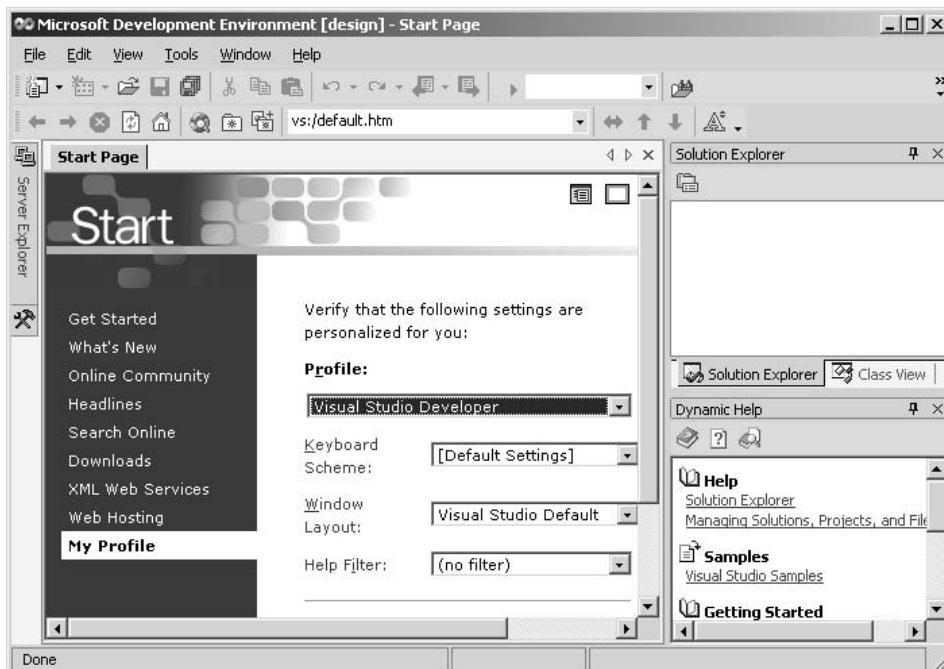
The Visual Studio .NET is an IDE (Integrated Development Environment) toolkit that makes .NET development much easier and much more productive. It is a unified-shared toolkit that there is only one environment to configure and to use regardless of the types of the programming languages. It simplifies developments of scalable Windows and Web .NET components. The Visual Studio .NET provides GUI interface to access all services available in all layers of the .NET Framework. Most of command line works can be undertaken here visually.

The Visual Studio .NET supports a variety of project types that you might undertake, including Windows applications, Windows service, Class library(DLL), Control library for Windows Form, ASP .NET Web application for Web site, ASP .NET Web Service for XML SOAP interface, Console applications, and other projects.

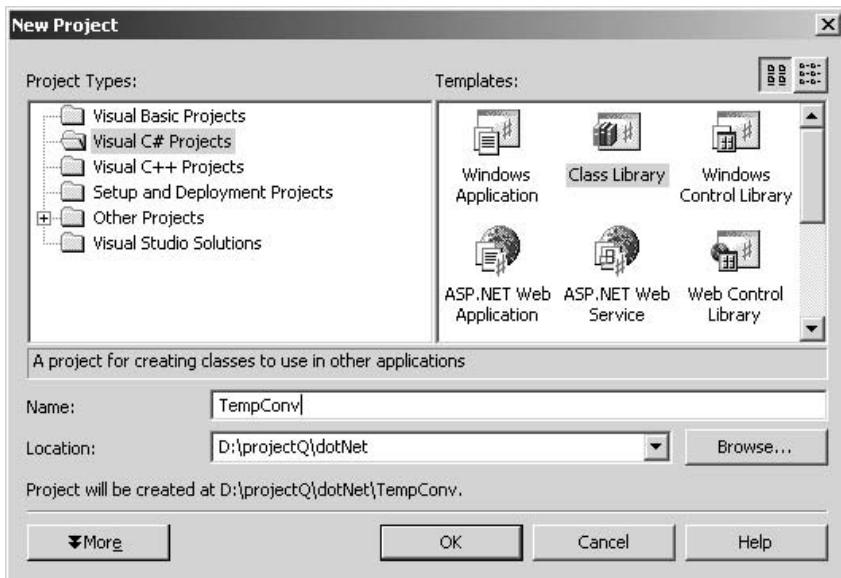
We focus the discussion on .NET component development by Visual Studio .NET. First, we show the development of a .NET temperature conversion component we discussed before. Then, the development of two clients of these components is demonstrated in detail, one in Windows Form and the others in Web Form developed in ASP .NET and deployed on MS IIS Web server to be accessed by any Web browser. Both of them reuse a common component to convert the temperature between Fahrenheit and Celsius degree.

6.5.1 Build a .NET Component

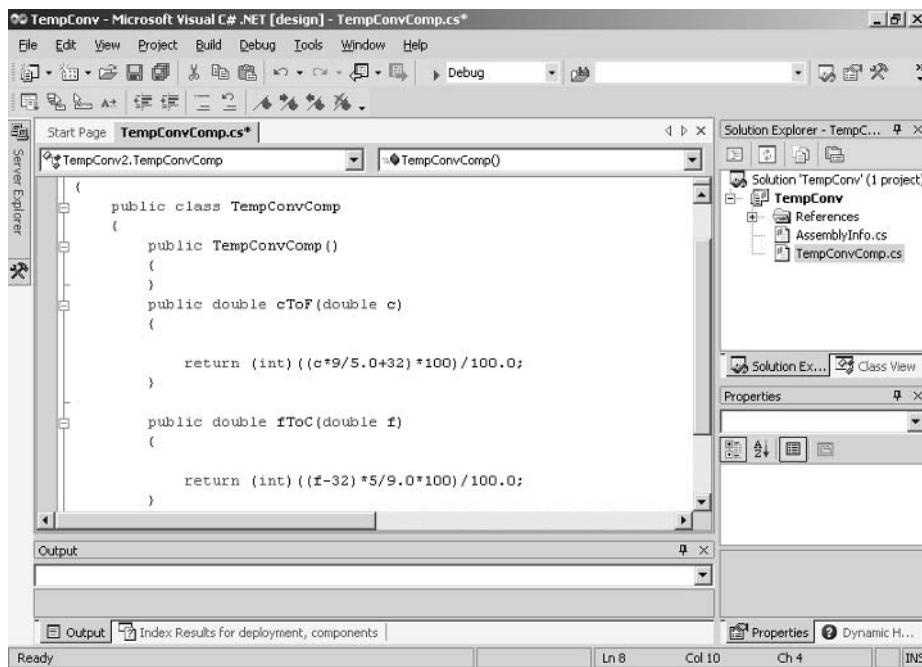
Step 1: Start up Visual Studio .NET and configure your profile. The default setting works for most of the cases unless any specific customization is needed.



Step 2: Create a new project by selecting “Visual C# Projects” with a template “Class Library” that will result in a DLL component.

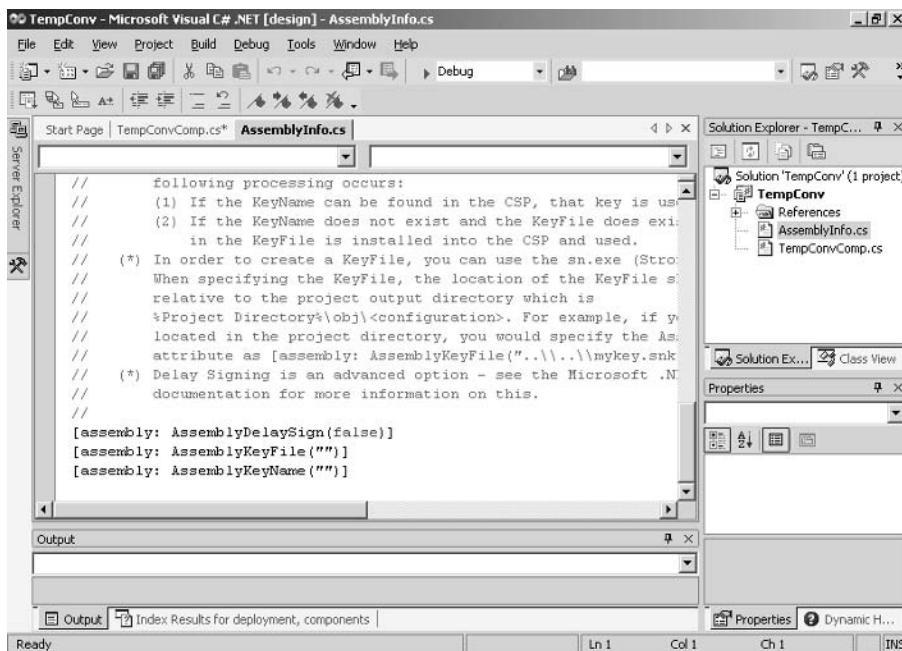


Step 3: Add in the source code for this TempConvComp component class.

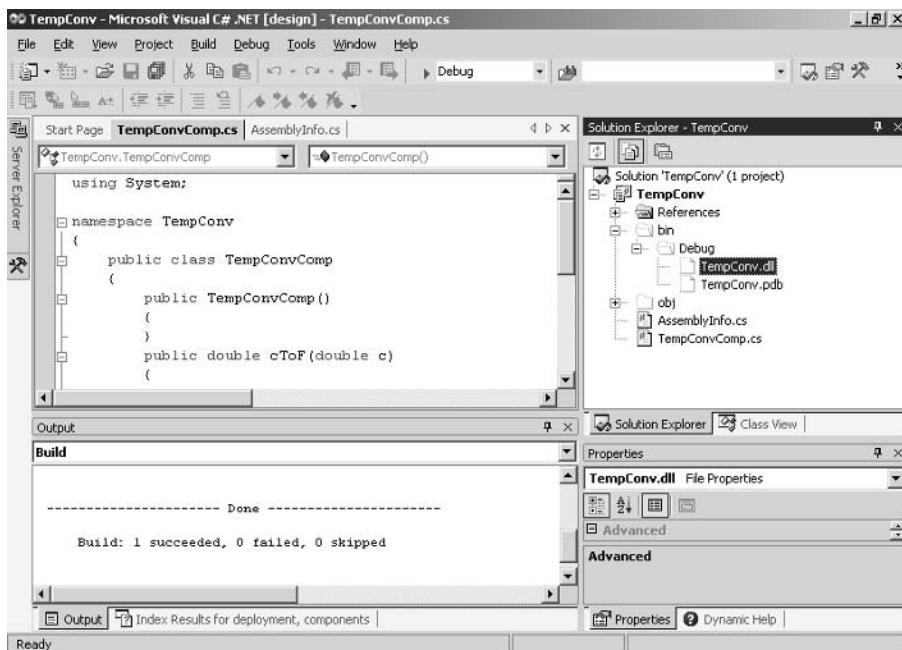


Step 4: Build and deploy the component.

Build the component by choosing the “Build” option on the top menu bar and configure the AssemblyInfo.cs file created by Visual Studio if a shared component deployment is necessary.

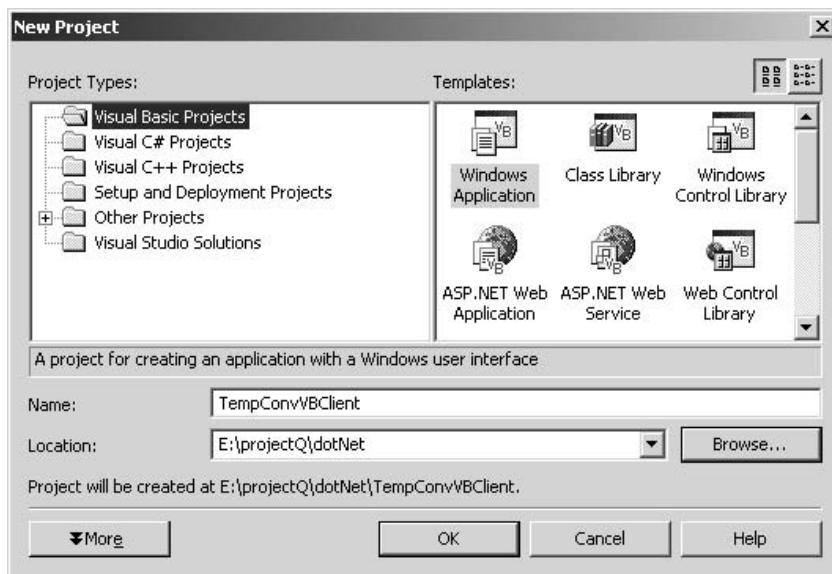


Finally, you will see that a DLL component is created as shown in the following:

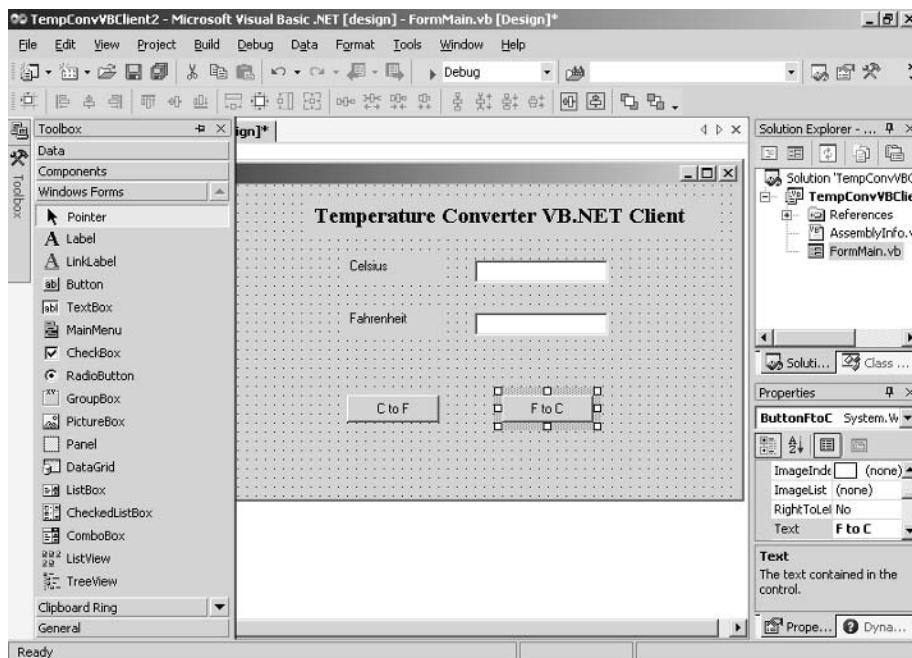


6.5.2 Build a Windows Form Client to the .NET Component

Step 1: Create a windows application in a Visual Basic Project.

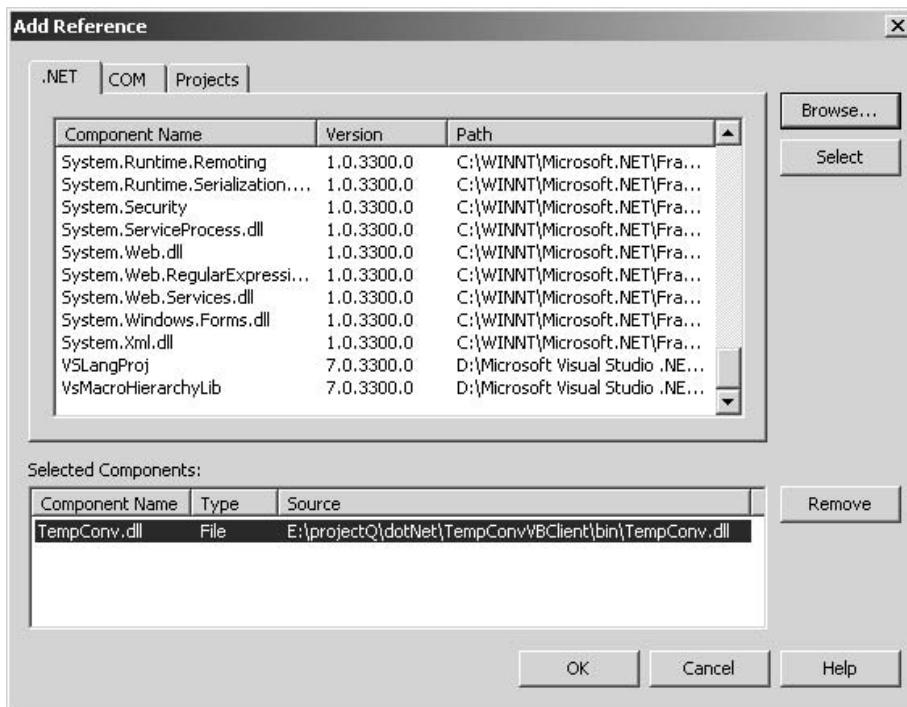


Step 2: Design a GUI interface for this Windows Form by Dragging and Dropping the GUI components from the Toolbox on the left to the design form.



Step 3: Add a reference to the .NET component “TempConv.dll” in this client.

Select the menu item “Projects” from the top bar menu or right click on the references in the “Solution Explorer” panel to specify the name of the component to be used.



Step 4: Build the client.

The source code of this Windows Form Client is shown as follows:

```

Public Class Form1
    Inherits System.Windows.Forms.Form
    Private Sub ButtonFtoC_Click(ByVal sender As
        System.Object, ByVal e As System.EventArgs) Handles
        ButtonFtoC.Click
        Dim objConv As New TempConv.TempConvComp()
        Dim f As Double
        f = CDbl(TextBoxF.Text)
        TextBoxC.Text = objConv.fToC(f)

    End Sub

    Private Sub ButtonCtoF_Click(ByVal sender As System.Object,
        ByVal e As
        System.EventArgs) Handles ButtonCtoF.Click
        Dim objConv As New TempConv.TempConvComp()
        Dim c As Double
    End Sub

```

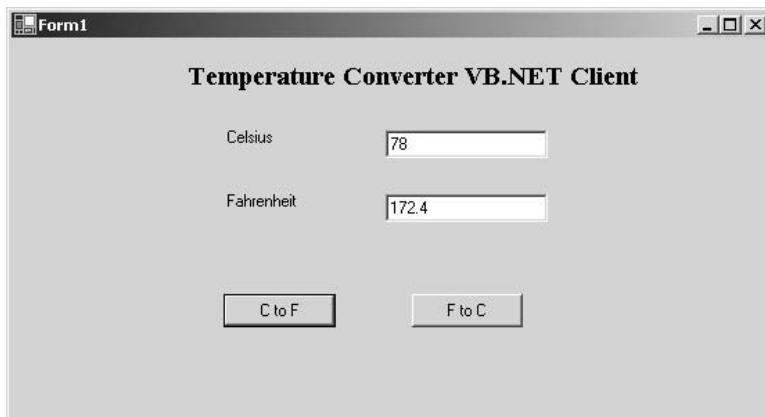
```

c = CDbl(TextBoxC.Text)
TextBoxF.Text = objConv.cToF(c)

End Sub
End Class

```

The next diagram shows the result of the execution of this VB .NET client to a CS .NET component.

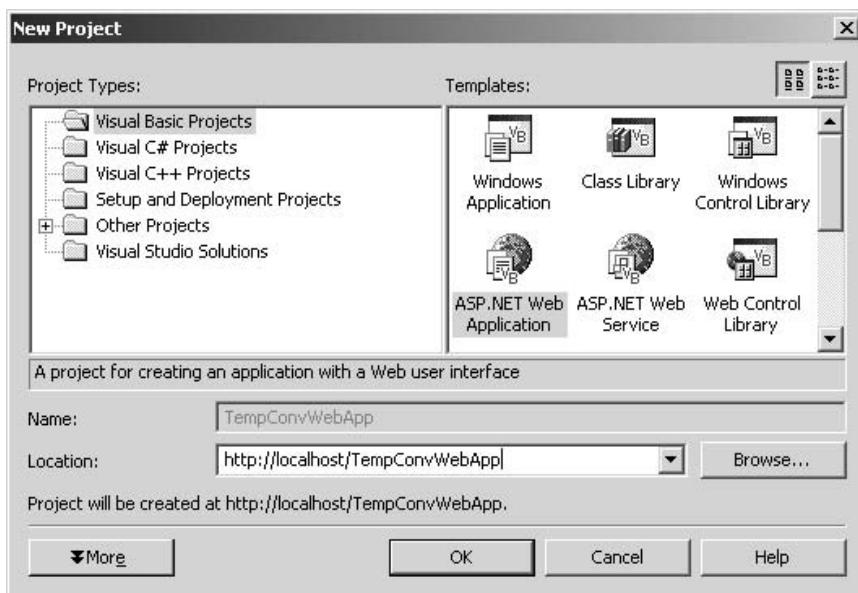


6.5.3 Build a Web Form Client for This .NET Component

Step 1: Make sure IIS is installed before you run Web application.

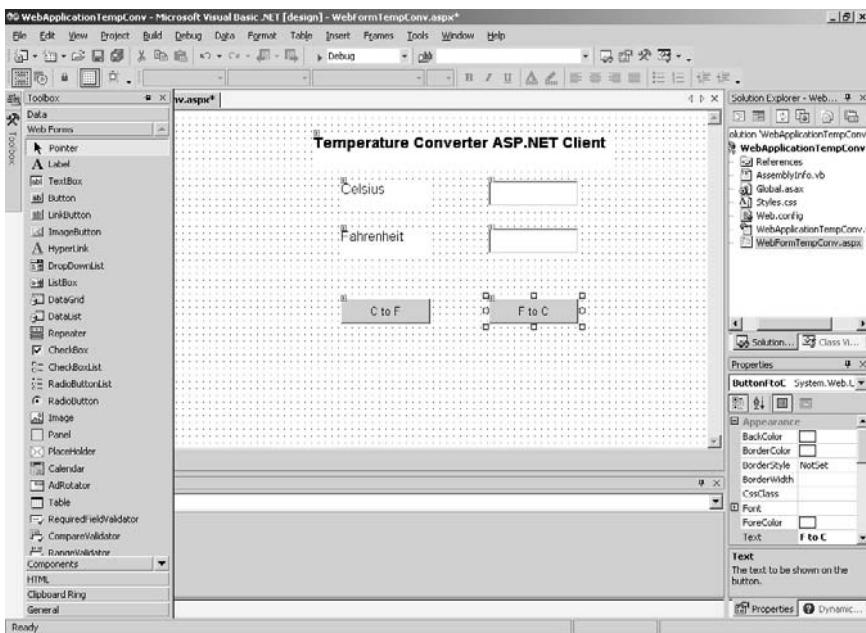
Your Web application will be saved to C:\Inetpub\wwwroot by default. The virtual directory *localhost* will be matched to this physical directory.

Step 2: Create an ASP.NET Web Application in a VB .NET project.

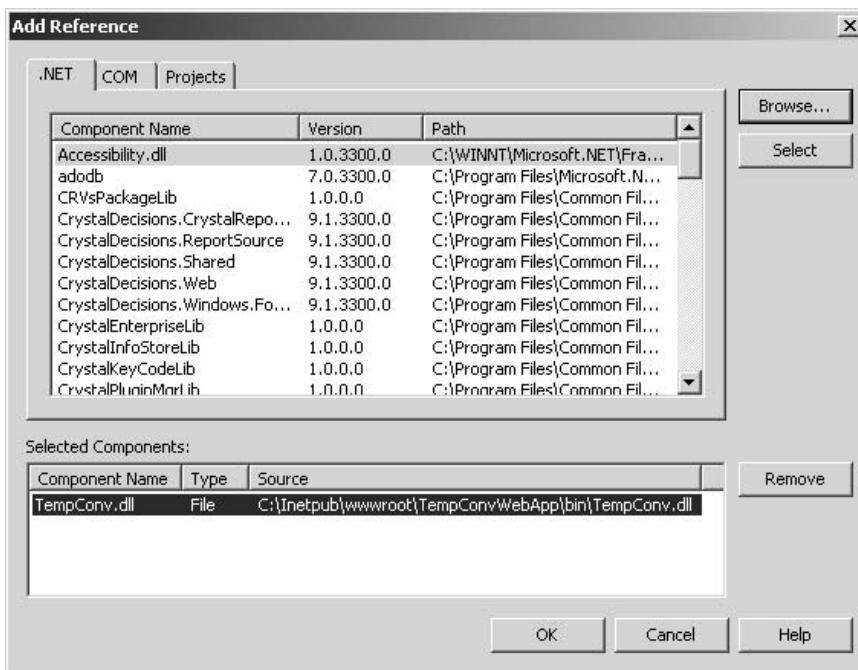


Step 3: Design GUI interface in this Web Form.

The process is the same as the GUI design in Windows Form.



Step 4: Bind This Web Form with the .NET component by “Add reference,” which is TempCov.dll. “Add Reference” option is available in the menu item “Project” on the top menu bar.



The source code of this Web Form client is shown as follows:

```

Public Class WebForm1
    Inherits System.Web.UI.Page
    Protected WithEvents Label1 As
        System.Web.UI.WebControls.Label
    Protected WithEvents Label2 As
        System.Web.UI.WebControls.Label
    Protected WithEvents Label3 As
        System.Web.UI.WebControls.Label
    Protected WithEvents Button1 As
        System.Web.UI.WebControls.Button
    Protected WithEvents TextBoxC As
        System.Web.UI.WebControls.TextBox
    Protected WithEvents TextBoxF As
        System.Web.UI.WebControls.TextBox
    Protected WithEvents Button2 As
        System.Web.UI.WebControls.Button
    Private Sub Page_Load(ByVal sender As System.Object, ByVal
        e As System.EventArgs) Handles MyBase.Load
        'Put user code to initialize the page here
    End Sub

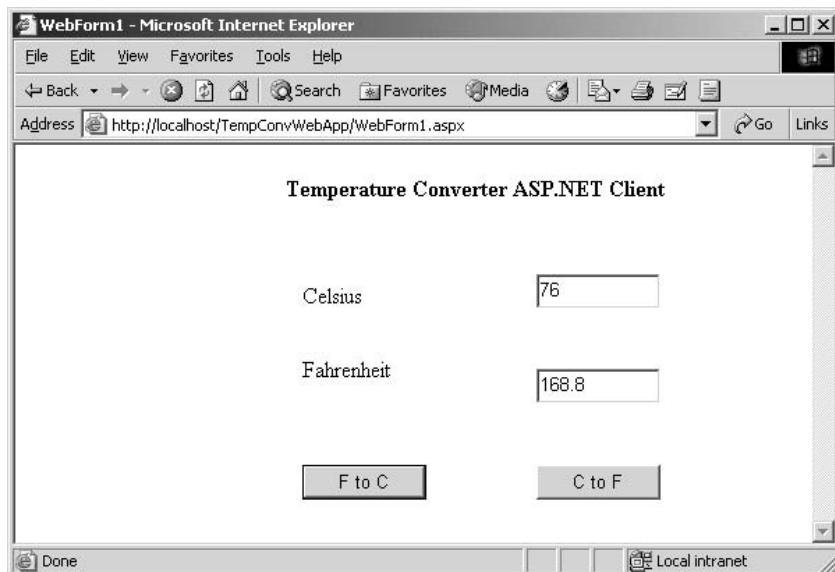
    Private Sub Button1_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles Button1.Click
        Dim objConv As New TempConv.TempConvComp()
        Dim f As Double
        f = CDbl(TextBoxF.Text)
        TextBoxC.Text = objConv.fToC(f)
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles Button2.Click
        Dim objConv As New TempConv.TempConvComp()
        Dim c As Double
        c = CDbl(TextBoxC.Text)
        TextBoxF.Text = objConv.cToF(c)
    End Sub
End Class

```

Step 5: Build the Web Form client for the .NET component.

Press the Build menu item and take an action to produce this Web Form, which is saved as an `.aspx` file that can be browsed by any internet browser. The `WebForm1.aspx` can be browsed just like a HTML page but it is an interactive page. The following picture shows this `.aspx` file browsed by Internet Explorer.



6.6 EXAMPLES AND LAB PRACTICE

This section is designed to enhance the understanding of .NET component concepts by providing some concrete examples and step-by-step guidelines that demonstrate how to build different types of .NET components, how to assemble and deploy the components, and how to build a client for these components and run the clients.

This section focuses on the development by command lines of .NET framework SDK. Here are four labs:

Lab 1 describes the steps to build a C# private component with a managed C++ client. It also demonstrates the running result of this console application.

Lab 2 shows the steps to build a C# client for the same component and execution result of this console application.

Lab 3 depicts the deployment of a shared .NET component.

Lab 4 illustrates the detailed steps to build up a distributed .NET component and its remote .NET client. The server-distributed component is running at a different process from the remote client, which is running in another process.

All source code in these labs are available in the attached companion CDROM.

6.6.1 Lab 1: Build a Private Component in C# .NET (Client Using Managed C++)

Step 1: We repeat the code one more time here, just for convenience.

```
using System;

namespace TempConv
{
```

```

public class TempConvComp
{
    public TempConvComp()
    {
    }

    public double cToF(double c)
    {
        return (int)((c*9/5.0+32)*100)/100.0;
    }

    public double fToC(double f)
    {
        return (int)((f-32)*5/9.0*100)/100.0;
    }
}
}

```

Step 2: Build a .NET DLL component.

```
>csc.exe /t:library /debug+ /out:TempConv.dll TempConvComp.cs
```

This component will be referenced by Lab 1 and Lab 2; you can copy the dll file to the corresponding project fold to be used.

Step 3: Develop client source code.

```

#using <mscorlib.dll>
using namespace System;
#using "TempConv.dll"

void main()
{
    double choice;
    double input;
    double output;
    bool next = true;

    TempConv::TempConvComp *myCSConvComp = new
    TempConv::TempConvComp();

    while (next) {
        Console::WriteLine(S"Please enter your choice:
        1 - Converter from F to C,
        2 - from C to F,
        3 - exit");
        choice=Double::Parse (Console::ReadLine());
        if (choice == 1) {
            Console::WriteLine(S"Please tell the temperature in F: ");
            input=Double::Parse (Console::ReadLine());
            output = myCSConvComp->fToC(input);
            Console::WriteLine(output);
        }
    }
}

```

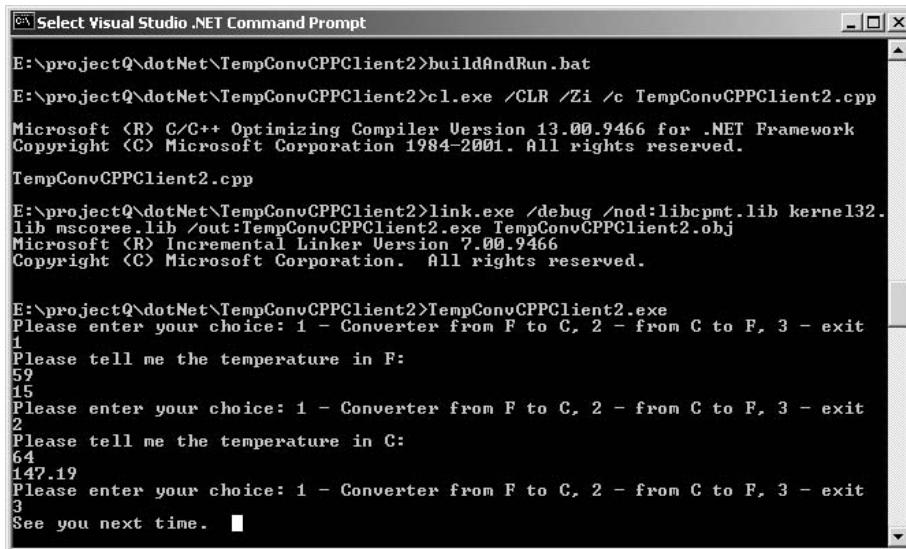
```
    else if (choice ==2){  
        Console::WriteLine(S"Please tell the temperature in C: ");  
        input=Double::Parse(Console::ReadLine());  
        output = myCSConvComp->cToF(input);  
        Console::WriteLine(output);  
    }  
    else {  
        next= false;  
        Console::WriteLine (S"See you next time.");  
    }  
}
```

Step 4: Build a C++ client who uses the .NET component and run it.

All the following commands are saved in a batch file called `BuildAndRun.bat` file.

```
>cl.exe /CLR /Zi /c TempConvCPPClient2.cpp  
>link.exe /debug /nod:libcpmt.lib kernel32.lib mscoree.lib  
/out:TempConvCPPClient2.exe TempConvCPPClient2.obj  
>TempConvCPPClient2.exe
```

The following picture shows the output of the execution of this batch file.



6.6.2 Lab 2: Build a Console Application with a C# Client Accessing .NET Component

Step 1: Develop source code for the C# client.

```

using System;
using TempConv;

class MainApp
{
    public static void Main()
    {
        TempConv.TempConvComp myCSTempConvComp = new
        TempConv.TempConvComp();
        double choice;
        double input;
        double output;
        bool next = true;

        while (next)
        {Console.WriteLine("Please enter your choice:
            1 - Converter from F to C,
            2 - from C to F,
            3 - exit");
        choice=Double.Parse (Console.ReadLine());
        if (choice == 1)
        {Console.WriteLine("Please tell the temperature in
            F: ");
        input=Double.Parse (Console.ReadLine());
        output = myCSTempConvComp.fToC(input);
        Console.WriteLine(output);
        }
        else if (choice ==2)
        {Console.WriteLine("Please tell the temperature in
            C: ");
        input=Double.Parse (Console.ReadLine());
        output = myCSTempConvComp.cToF(input);
        Console.WriteLine(output);
        }
        else
        {next= false;
        Console.WriteLine ("See you next time.");
        }
        }
    }
}

```

Step 2: Build a C# client accessing TempConv component. Remember to copy the DLL file into the same directory of the client because the component is a private component.

```

>csc /debug+ /reference:TempConv.dll /out:TempConvCSClient.exe
>TempConvCSClient.cs
>TempConvCSClient.exe

```

After we run this batch file “BuildAndRun.bat,” we will see the following output on the screen.

```
E:\projectQ\dotNet\TempConvCSClient2>buildAndRun.bat
E:\projectQ\dotNet\TempConvCSClient2>csc /debug+ /reference:TempConv.dll /out:TempConvCSClient.exe TempConvCSClient.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

E:\projectQ\dotNet\TempConvCSClient2>TempConvCSClient.exe
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
2
Please tell me the temperature in C:
98
208.4
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
1
Please tell me the temperature in F:
56.7
13.72
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
3
See you next time. ■
```

6.6.3 Lab 3: Build a Shared .NET Component by .NET Framework SDK

Step 1: Modify the source file to add two lines in [] in bold or make an AssemblyInfo file and put these lines in this file.

```
using System;
using System.Reflection;
[assembly:AssemblyVersion("1.0.0.0")]
[assembly:AssemblyKeyFile("originator.key")]

namespace TempConv
{
    public class TempConvComp
    {
        public TempConvComp()
        {
        }

        public double cToF(double c)
        {
            return (int)((c*9/5.0+32)*100)/100.0;
        }

        public double fToC(double f)
        {
            return (int)((f-32)*5/9.0*100)/100.0;
        }
    }
}
```

Step 2: Develop client source code.

```

using System;
using TempConv;

class MainApp
{
    public static void Main()
    {
        TempConv.TempConvComp myCSTempConvComp = new
        TempConv.TempConvComp();
        double choice;
        double input;
        double output;
        bool next = true;

        while (next)
        {
            Console.WriteLine("Please enter your choice:
                1 - Converter from F to C,
                2 - from C to F,
                3 - exit");
            choice=Double.Parse (Console.ReadLine());
            if (choice == 1)
            {
                Console.WriteLine("Please tell the temperature in F: ");
                input=Double.Parse (Console.ReadLine());
                output = myCSTempConvComp.fToC(input);
                Console.WriteLine(output);
            }
            else if (choice ==2)
            {
                Console.WriteLine("Please tell me the temperature in C: ");
                input=Double.Parse (Console.ReadLine());
                output = myCSTempConvComp.cToF(input);
                Console.WriteLine(output);
            }
            else
            {
                next= false;
                Console.WriteLine ("See you next time.");
            }
        }
    }
}

```

Step 3: Build a shared .NET component and its client, run it.

Create a pair of public/private key by **sn** command.

Compile the client source code which references the component. DLL file.

Register the built component with the GAC to be a shared component.
Run the client.

```
>sn -k originator.key
>csc /t:library /out:TempConv.dll TempConvComp.cs
>gacutil /i TempConv.dll
>csc /r:TempConv.dll /t:exe /out:TempConvCSClient.exe
  >TempConvCSClient.cs
>TempConvCSClient.exe
```

All these commands are saved in “BuildAndRun.bat” batch file. The next screen shows the result from this batch file.

```
c:\Select Visual Studio .NET Command Prompt
E:\projectQ\dotNet\sharedComponent>build
E:\projectQ\dotNet\sharedComponent>sn -k originator.key
Microsoft (R) .NET Framework Strong Name Utility Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Key pair written to originator.key

E:\projectQ\dotNet\sharedComponent>csc /t:library /out:TempConv.dll TempConvComp.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

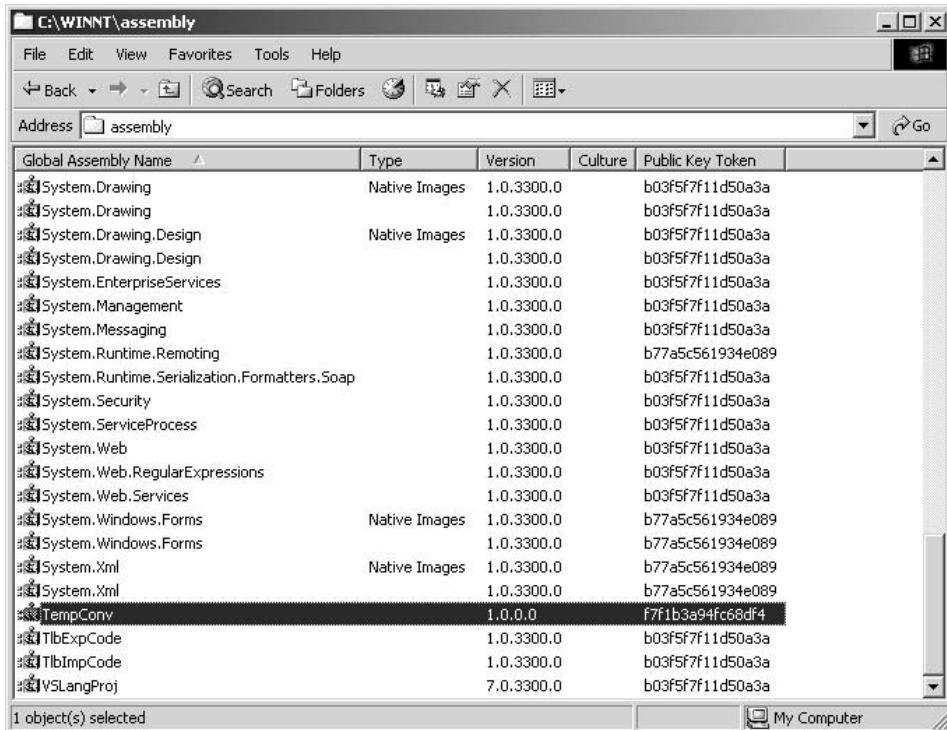
E:\projectQ\dotNet\sharedComponent>gacutil /i TempConv.dll
Microsoft (R) .NET Global Assembly Cache Utility Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Assembly successfully added to the cache

E:\projectQ\dotNet\sharedComponent>csc /r:TempConv.dll /t:exe /out:TempConvCSClient.exe TempConvCSClient.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

E:\projectQ\dotNet\sharedComponent>TempConvCSClient.exe
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
1
Please tell me the temperature in F:
56
13.33
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
2
Please tell me the temperature in C:
78
172.4
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
3
See you next time.
```

After the shared component deployment, this shared component can be found in assembly subdirectory of WINNT directory as shown in the following:



6.6.4 Lab 4: Distributed .NET Components by .NET Framework SDK

Step 1: Develop the source code for the distributed remote component.

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class CoTempConv : MarshalByRefObject
{
    public static void Main()
    {
        TcpChannel channel = new TcpChannel(4000);
        ChannelServices.RegisterChannel(channel);

        RemotingConfiguration.RegisterWellKnownServiceType (
            typeof(CoTempConv), "TempConvCompDotNet",
            WellKnownObjectMode.Singleton);
        System.Console.WriteLine("Hit <enter> to exit...");
        System.Console.ReadLine();
    }

    public double cToF(double c)
    {
        return (int)((c*9/5.0+32)*100)/100.0;
    }
}

```

```

        }

    public double fToC(double f)
    {
        return (int)((f-32)*5/9.0*100)/100.0;
    }
}

```

Step 2: Develop the source code for the remote client that accesses the distributed remote component.

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

class MainApp
{
    public static void Main()
    {
        try
        {
            TcpChannel channel = new TcpChannel();
            ChannelServices.RegisterChannel(channel);
            CoTempConv myCSTempConvComp = (CoTempConv)
                Activator.GetObject(typeof(CoTempConv),
                    "tcp://127.0.0.1:4000/
                    TempConvCompDotNet" );
            double choice;
            double input;
            double output;
            bool next = true;

            while (next)
            {
                Console.WriteLine("Please enter your choice:
                    1 - Converter from F to C,
                    2 - from C to F,
                    3 - exit");
                choice=Double.Parse (Console.ReadLine());
                if (choice == 1)
                {
                    Console.WriteLine("Please input the temperature in
                        F: ");
                    input=Double.Parse (Console.ReadLine());
                    output = myCSTempConvComp.fToC(input);
                    Console.WriteLine(output);
                }
                else if (choice ==2)
                {

```

```
Console.WriteLine("Please tell me the temperature  
    in C: ");  
input=Double.Parse (Console.ReadLine());  
output = myCSTempConvComp.cToF(input);  
Console.WriteLine(output);  
}  
else  
{  
next= false;  
Console.WriteLine ("See you next time.");  
}  
}  
}  
}  
}  
}  
catch (Exception e)  
{  
Console.WriteLine(e.ToString());  
}  
}  
}
```

Step 3: Build the distributed server component and its client component.

```
>csc TempConvComp.cs  
>csc /r:TempConvComp.exe TempConvCSClient.cs
```

Step 4: Run the server.

>TempconvComp.exe

```
E:\projectQ\dotNet\distributedComponent>buildServerAndClient.bat

E:\projectQ\dotNet\distributedComponent>csc TempConvComp.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

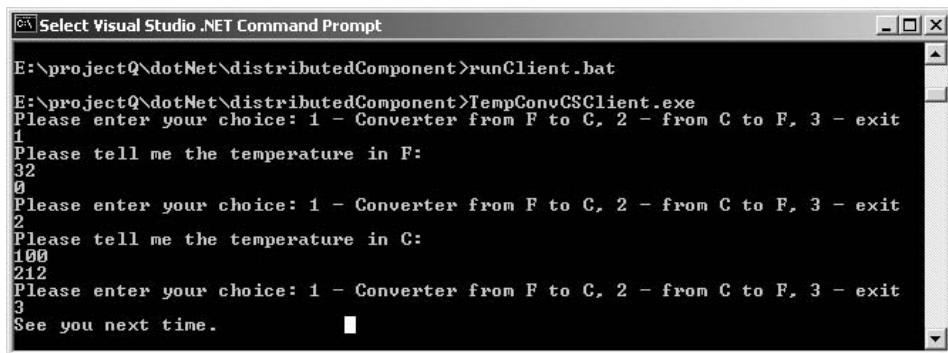
E:\projectQ\dotNet\distributedComponent>csc /r:TempConvComp.exe TempConvCSClient
.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

E:\projectQ\dotNet\distributedComponent>runServer.bat

E:\projectQ\dotNet\distributedComponent>TempconvComp.exe
Hit <enter> to exit...
```

Step 5: Run the remote client in a separate process.

>TempConvCSClient.exe



```
E:\projectQ\dotNet\distributedComponent>runClient.bat
E:\projectQ\dotNet\distributedComponent>TempConvCSCClient.exe
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
1
Please tell me the temperature in F:
32
0
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
2
Please tell me the temperature in C:
100
212
Please enter your choice: 1 - Converter from F to C, 2 - from C to F, 3 - exit
3
See you next time.
```

6.7 SUMMARY

The .NET component technologies are discussed in this chapter. The .NET framework and its constructs are introduced first. The .NET component is a precompiled MSIL file deployed as an assembly file that may contain multiple modules and resources self-described by its manifest. A .NET component can be deployed as a private component or a public shared component with strong name and GAC registration. The association of two .NET components can be composed by aggregation or containment. The invocation of method of one component from another component can be synchronous or asynchronous. A .NET component can be a local component accessed by other components within the same application domain or can be a distributed component accessed by other components remotely via Remoting channel.

6.8 SELF-REVIEW QUESTIONS

1. .NET component and its client
 - a. must be written in same programming language
 - b. may be written in different languages
 - c. may be running in different platforms
2. .NET component is deployed in
 - a. MSIL
 - b. bytecode
 - c. executable target code
3. .NET assembly may consist of
 - a. metadata
 - b. MSIL
 - c. resource files
 - d. modules
 - e. all

4. A .NET component has an extension
 - a. .dll
 - b. .exe
 - c. .netmodule
 - d. a or b
 - e. all
5. A .NET component itself
 - a. must be coded by a single programming language
 - b. may be assembled from many different modules in different languages
6. A .NET distributed component and its client must be run
 - a. in different machines
 - b. in the same machine
 - c. in the same application domain
 - d. in different application domains
7. A shared component must be registered
 - a. with NT Registry
 - b. with GAC
8. .NET component Versioning control is supported
 - a. .NET shared component
 - b. .NET private component
 - c. both of a and b
 - d. DLL COM
9. .NET delegate supports
 - a. synchronous event handler method invocation with event triggering
 - b. asynchronous event handler method invocation with event triggering
 - c. both
 - d. neither
10. .NET component composition can be completed by
 - a. containment
 - b. aggregation
 - c. both

Keys to Self-Review Questions

1. b 2. a 3. e 4. d 5. a 6. d 7. b 8. a 9. c 10. c

6.9 EXERCISES

1. What is .NET framework?

2. How is .NET framework class library organized?
3. How does .NET support?
4. What is the purpose of CLR?
5. What is the difference between CLR and JVM?
6. What is the difference between MSIL and JAVA bytecode?
7. Is .NET framework a cross-language platform?
8. How do you composite a .NET component nested inside another component?
9. Does private .NET component support side-by-side versioning?
10. Where is .NET component deployed?
11. Why is GAC needed for .NET shared component?
12. What is asynchronous delegate?
13. What is synchronous delegate?
14. When is class channel needed?
15. What is .NET module?
16. What is .NET assembly?
17. Is a module loadable?
18. Does a module have manifest?
19. Is a private key generated by command “SN” saved in manifest of component?
20. What is .NET application domain? Can a component access another component locally in another application domain?
21. Can an EXE file be deployed using .NET component?

6.10 PROGRAMMING EXERCISES

1. Design a .NET component in .NET C#, managed C#, .NET VB that provides services of a calculator. The calculator has the functionality to perform addition, subtraction, multiplication, and division of two real numbers. It can also detect the zero divisor in division operation. Use namespace in this component.
2. Design a client of this calculator in .NET VB, .NET C#, and .NET managed C#.
3. Deploy the server component “Calculator” as a private component.
4. Deploy the server component “Calculator” as a shared public component.
5. Rewrite component “Calculator” as a distributed .NET component.
6. Rewrite the client of this component to reuse the distributed component by Remoting Channel.
7. Plug in this “Calculator” component into a Window Form application.

8. Plug in this “Calculator” component into a Web form by ASP.NET and IIS to enable a user to browse it on-line.
9. View the manifest of assembly for component “Calculator.”
10. Design a client program using asynchronous delegate to let a server component to call back when the server component completes the request from a client.

REFERENCES

- [Chappel 2002] Chappel, David. *Understanding .NET*, Addison-Wesley, 2002.
[Code 2004] <http://www.codeproject.com>, 2004.
[Component 2004] <http://www.componentsource.com>, 2004.
[Csharp 2004] <http://www.csharphelp.com>, 2004.
[Lowy 2003] Lowy, J. *Programming .NET Component*, O'Reilly, 2003.
[MSDN 2004] <http://msdn.microsoft.com/library>, 2004.
[Platt 2003] Platt, David. *Introducing Microsoft .NET*, Microsoft Press, 2003.

7

COP WITH OSGi COMPONENTS

Objectives of This Chapter

- Introduce OSGi component architecture
- Discuss the infrastructures of OSGi technology
- Provide definitions of software components in OSGi
- Provide definitions of connectors for component assembling in OSGi
- Provide definitions of deployment model in OSGi

7.1 OVERVIEW OF OSGi TECHNOLOGY

OSGi stands for *Open Service Gateway initiative*, an industry alliance formed to specify, develop, and promote an open service platform for the delivery and management of multiple applications and services to all types of network devices in home, vehicle, mobile computing, and other environments [Chen 2002; OSGi 2003]. The OSGi specification defines a service platform, supporting service delivery and management. The first version of OSGi (version 1.0) specification was released in May 2000. The latest version (3.0) was released in April 2003. The OSGi specification has been designed to support a wide range of existing networking and computer technologies, enhancing existing wired and wireless networks and high-speed access technologies. The OSGi service platform includes a service gateway, a central device to enable, consolidate, and manage service request and service delivery between the local area network and the wide area network, such as the Internet. The service gateway can also function as an application server for high-level services such as healthcare services and security

monitoring, appliance monitoring and control in a home environment, or GPS-enabled tour guide, and emission detection and control in a vehicle environment.

The OSGi framework serves as the common environment for service components called *bundles*, which are software components plugged into the framework dynamically and provide various services. A service performs actual business task in the OSGi world. For instance, a service may monitor or control home temperature, obtain a customized stock quotes or interactive TV programs, and even check the merchandise quantity in a remote vending machine. Technically, a service is a Java interface with defined semantics and potentially multiple implementations. Services are packaged along with their associated resources such as images, HTML pages, and other data files. The interface specifies what the service will do, while the implementation provides the details of how the service is performed. One service bundle is a reusable and self-contained software that can be downloaded into the gateway and executed in a plug-and-play manner. Service bundles are insulated from each other in that code within one bundle cannot refer to classes inside another bundle, nor instantiate them or invoke their methods. Therefore, service framework and bundles in OSGi specification provide a component infrastructure, where high-level service bundles act as software components in component-oriented programming [Brown 1998]. In the rest of this chapter, we will use OSGi model for the component architecture embodied in OSGi specification.

Service bundles interact with each other by requesting or providing services at runtime. One bundle connects to other bundles through the OSGi framework, and their connection declarations are defined in a manifest file. The framework provides a hosting environment with the following services:

- Managing the life cycle of bundles.
- Resolving interdependencies among bundles and making classes and resources available from a bundle.
- Maintaining a registry of services.
- Firing events and notifying listeners when a bundle's state changes, when a service is registered or unregistered, or when the framework is launched or raises an error.

There exist several implementations for different versions of OSGi specifications, to name just a few: Java Embedded Server (JES) from Sun Microsystems [Sun 2003], mBedded Servers from ProSyst Software [ProSyst 2003], and Oscar [Oscar 2003]. The rest of this chapter will discuss the component model, the connection model, and the deployment model in OSGi component infrastructure based on JES 2.0, which is available from [Sun 2003].

7.2 COMPONENT MODEL OF OSGI

The simplest bundle is one without service interface. It has an activator class and a manifest file. A bundle activator class is a Java class that implements `org.osgi.framework.BundleActivator` and defines a pair of methods: `start` and `stop`. After a bundle is installed, the JES framework will call the `start` and `stop` methods to start and stop the bundle. There are three steps to create a simplest bundle:

Step 1: Write a bundle activator class in Java.

Step 2: Create a **Manifest** file as a text file.

Step 3: Build a JAR file that contains the compiled activator class the **Manifest** file.

Below is an example of simplest bundle, called **Display.jar**, which simply displays a message when started:

```

1 package simplest;
2
3 import org.osgi.framework.*;
4
5 public class Display implements BundleActivator {
6
7     public void start(BundleContext ctxt) {
8         System.out.println("Hello", my name is Alice.'');
9     }
10
11    public void stop(BundleContext ctxt) {
12        System.out.println("Good bye, see you later.");
13    }
14 }
```

Code 7.1. File name: **Display.java**

Line 1 puts the Activator class in the package named **simplest**. Line 3 imports the classes from **org.osgi.framework** package, including **BundleContext** used in the method parameter and **BundleActivator** interface. Any bundle activator class must define its **start** and **stop** methods. In this example, both these methods simply display a piece of message.

Every bundle installed in the JES framework contains a **Manifest** file, a standard text file that describes the contents of the JAR file. The **Manifest** file is structured in headers, and each header has an attribute. Our example manifest file contains only one header as below:

Bundle-Activator: simplest.Display

Code 7.2. File name: **Manifest**

The headers in a manifest file tell the JES framework where to find resources within the bundle during the bundle life cycle. The OSGi specification defines a number of headers as shown in Table 7.1:

Now that we have a bundle activator class and a manifest file, we are ready to create a bundle. Suppose we have both **Display.java** and **Manifest** files created in our current working directory, and the JES **framework.jar** in our CLASSPATH, issue the following two commands:

```
javac -d . Display.java
jar cmf Manifest Display.jar simplest\*.class
```

TABLE 7.1. Headers for Bundle Manifest Files

Bundle-Activator	Bundle-DocURL	Bundle-Version
Bundle-ClassPath	Bundle-Name	Export-Package
Bundle-ContactAddress	Bundle-NativeCode	Export-Service
Bundle-Description	Bundle-UpdateLocation	Import-Package
Bundle-Vendor		Import-Service

This will create a bundle called `Display.jar` in our current-working directory. Now, start the JES server and install the `Display.jar` bundle. When we start and stop this bundle, we will see a printed message “Hello, my name is Alice” and “Good bye, see you later.” respectively.

As we discussed previously, bundles represent components in OSGi component architecture. Components are connected through component interface and services. Services are designed to be written with interface and implementation separated. The interface is exposed to other components, which acts as a contract between the client components and the service component. The separation of interface and its implementation makes it possible to have different implementations for one interface. An example with the separation of interface and implementation is given below:

```

1 package hello.service;
2
3 public interface HelloService {
4     public void hello(String firstName, String lastName,
5     String title);
6 }
```

Code 7.3. File name: `HelloService.java`

This interface is named `HelloService` (line 3). The `hello` method is declared in line 4 with three parameters: the individual’s first name, last name, and title. Any implementation of this service will provide a concrete way to say “hello.” Let us see a first implementation.

```

1 package hello.impl;
2
3 import java.util.*;
4 import hello.service.HelloService;
5
6 class HelloImpl implements HelloService {
7
8     public void hello(String firstName, String lastName,
9     String title) {
```

```

9     System.out.println("Hello, " + title + " " + firstName +
10    "+ lastName + "!");
11 }

```

Code 7.4. File name: HelloImpl.java

Notice that this implementation class is stored in a different package than the interface class (line 1). Line 6 defines the class to implement the `HelloService` interface. Notice that the implementation class has default package access, while the `HelloService` interface has public access. This means that other bundles can call the interface but not the implementation. In the `hello` method (line 8), this implementation provides a simple greeting method.

Next, we need to write a bundle activator class. In this example, we must register our new service, `HelloImpl`, so that other bundles can access this new service, and whenever this new service is requested, its `start` method will be called. In our bundle activator class, we will register services using the `registerService` method that the `BundleContext` object defines. If the service registration is successful, the framework returns a unique `ServiceRegistration` object to the bundle. The `ServiceRegistration` object lets other bundles get a reference to the service or update a service's properties. We must use the `ServiceRegistration` object to unregister, so only the bundle that holds the `ServiceRegistration` object can unregister the service.

```

1 package hello.impl;
2
3 import java.util.Properties;
4 import org.osgi.framework.*;
5 import hello.service.HelloService;
6
7 public class Activator implements BundleActivator {
8
9     private ServiceRegistration reg=null;
10
11     public void start(BundleContext ctxt) throws
12         BundleException {
13         HelloService hs = new HelloImpl();
14         Properties props = new Properties();
15         props.put("description", "hello");
16
17         reg = ctxt.registerService("hello.service.HelloService",
18             hs, props);
19
20         hs.hello("Andy", "Wang", "Dr.");
21         hs.hello("Don", "Carpenter", "Professor");
22         hs.hello("Ken", "Messersmith", "Mr.");
23     }
24
25     public void stop(BundleContext ctxt) throws
26         BundleException {

```

```

24         if (reg != null)
25             reg.unregister();
26     }
27 }
```

Code 7.5. File name: Activator.java

The bundle activator class is stored in `hello.impl` (line 1), the same package as the implementation class. We need to import `java.util.Properties` (line 3), because we will create a `Properties` object in order to register the service. We also need to import the `HelloService` interface (line 5), because we will register our service under its interface name. Unlike the implementation classes, the `BundleActivator` class is declared `public` (line 7), so that the JES framework can call it. Line 9 creates a reference, named `reg`, to a `ServiceRegistration` object. Line 12 creates an instance of our service and casts it to its interface type because we will register the service under its interface name. Line 13 creates a `Properties` object and gives it a key and a value in line 14. Once the service is registered, the framework will return a `ServiceRegistration` object that you can store in that object reference variable (line 16). Notice that in line 16, we use the fully qualified name for the interface `HelloService`, the instance of the service and the `Properties` object that we just created to register the service. In the `stop` method (line 23–26), we check for a valid `ServiceRegistration` object, then use the `ServiceRegistration` object to unregister the service with the `unregister` method.

Now we can offer our services for other bundles to use. To do this, we add an `Export-Package` header to the `Manifest` file, which names the Java packages that the bundle offers to share with other bundles. Because a bundle always contains a bundle activator class, the `Manifest` file must also have a `Bundle-Activator` header, as below:

```

Bundle-Activator: hello.impl.Activator
Export-Package: hello.service
```

Code 7.6. File name: Manifest

After compiling these source codes, we have the following file structure, starting with our current working directory:

```

Current-Working Directory
    Manifest (file)
    hello (directory)
        service (directory)
            HelloService.class (file)
        impl (directory)
            HelloImpl.class (file)
            Activator.class (file)
```

Issuing the following command from the current working directory

```
jar cmf Manifest Hello.jar hello\service\*.class
hello\impl\*.class
```

will create a bundle named `Hello.jar`. Install and start this bundle, and we will see the following display:

```
Hello, Dr. Andy Wang!
Hello, Professor Don Carpenter!
Hello, Mr. Ken Messersmith!
```

We can examine the registered services by issuing a command `services` at the JES server prompt and we will get the following:

```
> services
[hello.service.HelloService]
    description=hello
```

The property `description=hello` was set with the `Properties.put` method (line 14) in the source code `Activator.java`.

We could provide a different implementation of `HelloService` called `HelloImpl2.java` that is similar to `HelloImpl.java` except that the implementation of the `hello` method is different.

```
1 package hello2.impl;
2
3 import java.util.*;
4 import hello.service.HelloService;
5
6 class HelloImpl2 implements HelloService {
7
8     public void hello(String firstName, String lastName,
9                        String title) {
10         System.out.println("Hello, " + title + " " + firstName +
11                            " " + lastName + "!");
12         System.out.println("How are you? -- I'm fine, thank
13                           you.");
14     }
15 }
```

Code 7.7. File name: `HelloImpl2.java`

This `HelloImpl2.java` also imports and implements the `HelloService` interface (lines 4 and 6). The `hello` method is implemented as before, but this time it displays more greeting messages (lines 9 and 10). The new implementation must also have a new bundle activator class that is similar to the one before but that registers our new service, `HelloImpl2`.

```
1 package hello2.impl;
2
3 import java.util.Properties;
4 import org.osgi.framework.*;
```

```

5 import hello.service.HelloService;
6
7 public class Activator implements BundleActivator {
8
9     private ServiceRegistration reg =null;
10
11    public void start(BundleContext ctxt) throws
12        BundleException {
13        HelloService hs = new HelloImpl2();
14        Properties props = new Properties();
15        props.put("description", "hello2");
16
17        reg = ctxt.registerService("hello.service.HelloService",
18            hs, props);
19
20        hs.hello("Andy", "Wang", "Dr.");
21        hs.hello("Don", "Carpenter", "Professor");
22        hs.hello("Ken", "Messersmith", "Mr.");
23    }
24
25    public void stop(BundleContext ctxt) throws
26        BundleException {
27        if (reg != null)
28            reg.unregister();
29    }
30 }
```

Code 7.8. File name: Activator.java for a different implementation

The manifest file for this new implementation is listed below:

Bundle-Activator: hello2.impl.Activator
Export-Package: hello.service

Code 7.9. File name: Manifest2 for the new implementation

Notice that the Hello2 bundle exports the package hello.service, which includes HelloService.java. Recall that the bundle Hello exports the hello.service package as well. If another bundle wants to share hello.service, does it share with bundle Hello or bundle Hello2? The JES framework is designed so that the bundle that is started first will export the package. We can always check which bundle has exported a package using `exportedpackages` command from the JES framework command line.

After compiling these source codes, we have the following file structure, starting with our current working directory:

Current-Working Directory
Manifest2 (file)
hello2 (directory)
impl2 (directory)

```
HelloImpl2.class (file)
Activator.class (file)
```

Issuing the following command from the current working directory

```
jar cmf Manifest2 Hello2.jar hello\service\*.class
hello2\impl2\*.class
```

will create a bundle named `Hello2.jar`. Install and start this bundle, and we will see the following display:

```
Hello, Dr. Andy Wang!
How are you? -- I'm fine, thank you.
Hello, Professor Don Carpenter!
How are you? -- I'm fine, thank you.
Hello, Mr. Ken Messersmith!
How are you? -- I'm fine, thank you.
```

To summarize, a service should provide a public interface that specifies what it does and the necessary implementation classes that realize how it is done. The service must be registered with the framework to be useful for other bundles to access. The general process of developing a service bundle is listed below:

Step 1: Design the service interface.

Step 2: Implement the service.

Step 3: Write a bundle activator that usually registers the service in its `start` method and unregisters the service in its `stop` method.

Step 4: Declare the packages exported by the bundle in the `Export-Package` manifest header; the service interface should belong to the exported packages.

Step 5: Compile the classes and pack everything into a bundle JAR file.

We have discussed component definitions in Chapter 1 and we found out that it is not easy to have a unique component definition to fit in different situations. Such difficulties underlying the definition discussion are due to the following facts:

Software components are associated with their component infrastructure. Different component technologies have different component infrastructures, and thus have different component definitions.

In OSGi component infrastructure, a component is a packed JAR file, which is self-contained in that it contains class files and resources such as images, HTML pages, and other data files necessary for this component to fulfill its functions. Each OSGi bundle usually delivers at least one service that is reusable by other components. An OSGi component is not just a static archive file. It has a life cycle with state transitions in its lifetime, as illustrated in Figure 7.1.

When a component is developed and deployed, other components can use it through OSGi connection model discussed in the next section.

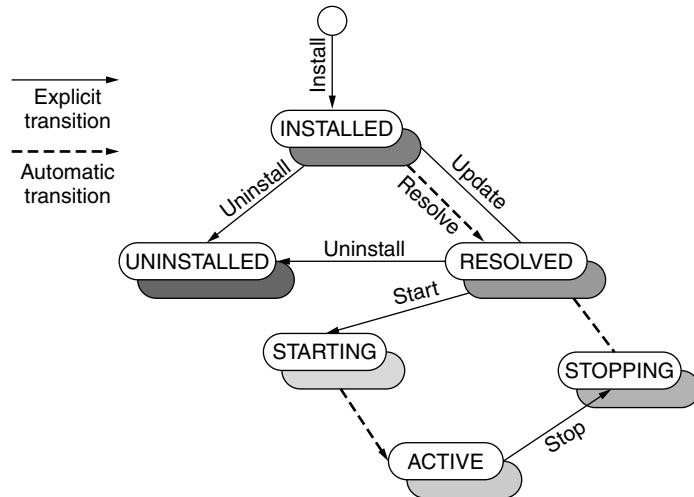


FIGURE 7.1. Bundle state transitions in its life cycle.

7.3 CONNECTION MODEL OF OSGi

There are two kinds of connections among OSGi components:

1. Static connections through import/export packages.
2. Dynamic connections through services.

Static connections define the architectural relationship among components in terms of their packages dependency. A package is a collection of classes that are grouped together according to their functionality. The Java API, for instance, consists of hundreds of such packages, forming a Java class library. Java uses a special keyword `package` to create packages at the first line of a Java source code. One OSGi component can *import* multiple packages, which indicates that the component needs to use some classes in the imported packages to fulfill its own tasks. On the other hand, one OSGi component can *export* packages, which indicates that the component is willing to make its packages available for other components to use. In object-oriented programming or library-based programming, one program can always import an API package or a library package without considering its availability at runtime. In component-oriented programming, however, one component has to import those packages that have been exported by other components so that its needs could be satisfied at runtime.

An OSGi component expresses its intention to export or import packages by declaring the packages with the `Export-Package` or `Import-Package` header respectively in its manifest file. For instance, the HTTP bundle in JES 2.0 has the following headers in its manifest file:

```
Export-Package: org.osgi.service.http; specification-version=1.0,
  com.sun.jes.service.http, com.sun.jes.service.ssl
Import-Package: javax.servlet; specification-version=2.1.1,
  javax.servlet.http; specification-version=2.1.1
```

Let diamonds represent packages and rectangles represent OSGi components, that is, OSGi bundles. Figure 7.2 illustrates that Bundle 1 exports package A and imports package B.

With this graphical notation, we can discuss the component connections in terms of their packages dependency. For example, Figure 7.3 demonstrates the package dependency among three bundles in JES 2.0: HTTP, Servlet, and Log.

The package dependency is determined at design time. Therefore, package dependency represents static component connections.

The second kind of component connection in OSGi component infrastructure is service connections. One OSGi component can provide certain services by registering its services with the framework. Once these services are registered, they can be accessed by other components. A component can provide multiple services if it desires. On the other hand, a component might not provide any service at all.

Suppose we have an OSGi component called FTP that provides file transfer services. In its manifest file, there is one line as the following:

```
Export-Package: ftp.service; specification-version=1.0
```

In order to register its service with the framework, FTP has a piece of code similar to the following in the `start()` method of its activator class:

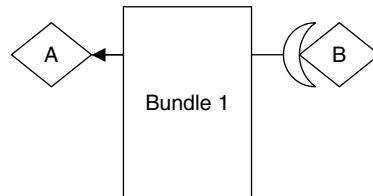


FIGURE 7.2. Export/import packages.

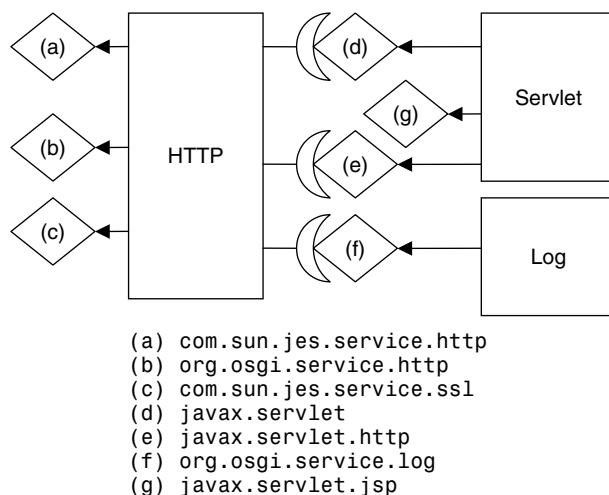


FIGURE 7.3. OSGi component connections via package dependency.

```

1 //in the FTP bundle's activator
2 public void start(BundleContext bctx) {
3     Properties props = new Properties();
4     props.put("port", new Integer(21));
5     FtpService ftp = new FtpServiceImpl();
6     bctx.registerService("ftp.service.FtpService", ftp,
7     props);
7 }
```

Code 7.10. Registering a service

The `BundleContext` in line 2 is a Java interface that provides bundle's execution context within the framework. It has a method called `registerService()` to register services in the framework's service registry. The first parameter to this method (line 6) is the class name under which the service can be located. The second parameter is the service object. The third parameter is the properties for this service. Note that the `Properties` object was created in line 3 and line 4. Since `java.util.Properties` inherits from `java.util.Hashtable` and again from `java.util.Dictionary`, `props` calls the `Hashtable` method `put()` to set a property by supplying a key and its value to the `put()` method (line 4).

After a service is registered, another component can use it through invoking the service's methods. Suppose a bundle called `Download` wants to use the file transfer service provided by `FTP` bundle. First of all, the manifest file of `Download` has to include an import header as below:

```
Import-Package: ftp.service; specification-version=1.0
```

Then, it could obtain such a file transfer service with the following code in the `start()` method of its activator class:

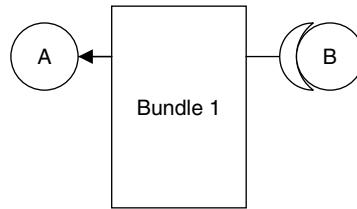
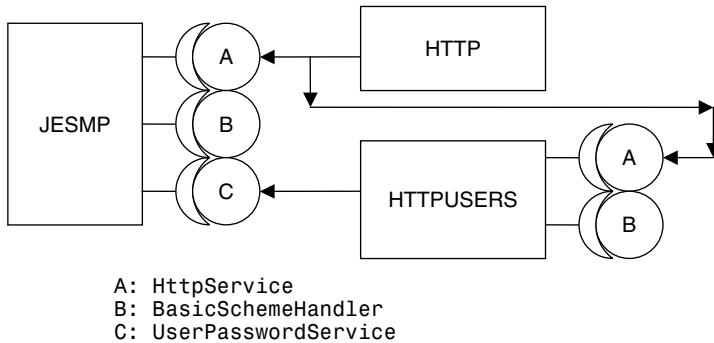
```

1 //in the Download bundle's activator
2 public void start(BundleContext bctx) throws Exception {
3     ServiceReference ref
4         =bctx.getServiceReference("ftp.service.FtpService");
5     FtpService ftp = (FtpService) bctx.getService(ref);
6     ftp.get(file);
6 }
```

Code 7.11. Using a service

In the code list 7.11, it calls `BundleContext`'s `getServiceReference` method in line 3, specifying the name of the service interface to get `ServiceReference` to the service. Then in line 4, it calls `BundleContext`'s `getService` method, passing in `ServiceReference` obtained in line 3, to get the service object. Finally, in line 5, file transfer is conducted by calling the method of the service object.

Let circles represent services and rectangles represent bundles. Figure 7.4 illustrates that Bundle 1 provides service A and requires service B.

**FIGURE 7.4.** OSGi services.**FIGURE 7.5.** OSGi service connection.

With this graphical notation, we can describe the service connections among the three components in JES 2.0 – JESMP, HTTP, and HTTPUSERS, as depicted in Figure 7.5.

In Figure 7.5, both JESMP and HTTPUSERS require the `HttpService` provided by the HTTP component. JESMP requires the `UserPasswordService` provided by HTTPUSERS. Both JESMP and HTTPUSERS require the `BasicSchemeHandler` service provided by another component called `HTTPAUTH` that is not displayed in the figure.

We have discussed two kinds of connections among OSGi components: package connections and service connections. Package connections represent static, architectural dependency among components, while service connections represent dynamic, runtime dependency among components. These two kinds of connections have a close relationship: package connections are the prerequisite for service connections. In other words, a server component must first export its service package and a client component must import the service package before the client can successfully establish a service connection with the server component.

7.4 DEPLOYMENT MODEL OF OSGi

An OSGi component is deployed by installing it in the OSGi framework. In JES 2.0, there is a default reusable component base located at the directory of `%JES_HOME%/bundles`. A bundle can be physically moved into this component base in order to be installed conveniently by referring its name only. Otherwise, you have to specify the complete URL for a component in order to install it.

An OSGi component is a Java archive (JAR) file called a bundle, which may reside on the local file system or on a remote server. The current OSGi specification does not provide specific mechanisms as how to find those bundles. The deployment of OSGi components requires that the location and the name of the bundles be known in advance.

Once a component has been installed, it will have unique identifier (ID) assigned by the framework, and its state becomes INSTALLED. Recall that a bundle has various states during its life cycle (Figure 7.1). Whenever a bundle enters one of the five states in {INSTALLED, STARTED, STOPPED, UPDATED, UNINSTALLED}, a bundle event will be generated to inform its listeners. Thus, a bundle can trigger five types of events, as summarized in Table 7.2.

After a bundle has been installed, the OSGi framework will examine its manifest file to see what packages this bundle imports/exports. If it needs to import packages, the framework checks whether those packages have been exported by other bundles. If yes, the bundle changes its state to RESOLVED and exports its packages. A user of the RESOLVED bundle can start the bundle by issuing a “start” command followed by the bundle ID in the OSGi framework. The framework responds to the user command by creating a `BundleContext` object for the bundle and calling the `start()` method of the bundle’s activator class. Then, the bundle registers its services, which is part of the `start()` method. This action generates a service event broadcast to interested listeners. There are three kinds of service events, as illustrated in Table 7.3.

TABLE 7.2. OSGi Bundle Events

Bundle Event	When the Event Is Fired	Listener Method Invoked
INSTALLED	A bundle has been installed in the framework.	<code>BundleListener.bundleChanged</code>
STARTED	A bundle has been activated.	<code>BundleListener.bundleChanged</code>
STOPPED	A bundle has been deactivated.	<code>BundleListener.bundleChanged</code>
UPDATED	A bundle has been updated.	<code>BundleListener.bundleChanged</code>
UNINSTALLED	A bundle has been uninstalled.	<code>BundleListener.bundleChanged</code>

TABLE 7.3. OSGi Service Events

Service Event	When the Event Is Fired	Listener Method Invoked
REGISTERED	A service has been registered.	<code>ServiceListener.serviceChanged</code>
UNREGISTERING	A service is being unregistered.	<code>ServiceListener.serviceChanged</code>
MODIFIED	The properties of a service have been modified.	<code>ServiceListener.serviceChanged</code>

When the `start()` method has been successfully executed and returned, the bundle changes its state from STARTING to ACTIVE. An active bundle is one running in the OSGi environment with all its dependency resolved.

A bundle can be updated at runtime, which allows a new version of the bundle to replace the old one. This is done by the `update` command followed by the bundle ID and the URL of the new version JAR file.

To stop a bundle, the OSGi framework calls the bundle activator's `stop()` method followed by a sequence of actions:

1. The framework unregisters the service provided by the bundle. This will trigger a service event broadcast to notify interested listeners that the service is being unregistered.
2. The framework releases any services in use by the bundle.
3. The framework removes any event listeners added by the bundle.
4. The bundle's state changes from ACTIVE to STOPPING and then to RESOLVED.
5. A bundle event is generated and broadcast to notify listeners that the bundle has been stopped.

Finally, a bundle can be uninstalled such that the bundle is removed from the OSGi framework environment. Again, a bundle event will be triggered to inform those listeners that the bundle has been uninstalled.

7.5 EXAMPLES AND LAB PRACTICE

This section discusses a few examples of component-oriented programming using OSGi component infrastructure. Readers are recommended to follow the steps with hands-on practice on a computer. As we discussed in previous sections, the OSGi component infrastructure defines service bundles as self-contained software components, and each component usually has a public interface and its implementation. The service interface specifies what this component does, and the service implementation class provides how this service is realized in detail. The process of developing an OSGi component has six steps as listed below:

- *Step 1:* Write a Java interface for the service bundle.
- *Step 2:* Write a Java class to implement the interface in Step 1.
- *Step 3:* Write a Java class, usually named as Activator, to implement the `BundleActivator` interface provided by `org.osgi.framework` package. The activator class usually registers the service in its `start()` method and unregisters the service in its `stop()` method.
- *Step 4:* Write a text file, usually named as `Manifest`, to define the bundle headers. The most important headers include: `Bundle-Activator`, `Export-Package`, and `Import-Package`.
- *Step 5:* Compile the Java source files and create a bundle JAR file.
- *Step 6:* Deploy the bundle JAR file to an OSGi server like JES 2.0 and start running it.

Example 7.1

Following the process given above, let us develop a weather service bundle that reports the current weather for a city.

Step 1: Write a Java interface for the weather service bundle.

```
package weather;
public interface WeatherService {
    public void getWeather(String location);
}
```

Code 7.12. File name: WeatherService.java

Step 2: Write a Java class to implement the interface.

```
package weather;
import weather.WeatherService;

class WeatherServiceImpl implements WeatherService {
    public void getWeather(String location) {
        System.out.println(location + " is sunshine, high
                           70, low 59");
    }
}
```

Code 7.13. File name: WeatherServiceImpl.java

Step 3: Write an activator class.

```
package weather;
import java.util.*;
import org.osgi.framework.*;
import weather.WeatherService;

public class Activator implements BundleActivator {
    private ServiceRegistration reg;

    public void start(BundleContext ctx) {
        WeatherService ws = new WeatherServiceImpl();
        Properties prop = new Properties();
        prop.put("description", "weather");

        reg = ctx.registerService("weather.WeatherService",
                                 ws, prop);
        ws.getWeather("Marietta");
    }

    public void stop(BundleContext ctx) {
```

```

        System.out.println("Thanks for using weather
                           service.");
        if (reg != null)
            reg.unregister();
    }
}

```

Code 7.14. File name: Activator.java*Step 4:* Write a text file named as **Manifest.txt**.

```

Export-Package: weather
Bundle-Activator: weather.Activator

```

Code 7.15. File name: Manifest.txt*Step 5:* Compile the Java source files and create a bundle JAR file with the following commands:

```

javac -d . -classpath %classpath%;c:\jes2.0\lib\framework.jar
      *.java
jar cmf Manifest.txt weather.jar weather\*.class

```

Step 6: Deploy the bundle JAR file to an OSGi server like JES 2.0 and start running it.**Example 7.2**

This example demonstrates a bundle that registers a servlet and an image included as a resource in the bundle.

Step 1: With a text editor, create the following source code files:

```

package servlet;
import org.osgi.framework.*;
import org.osgi.service.http.*;
import java.net.*;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class Activator implements BundleActivator {
    final static String IMAGE_ALIAS = "/example7_2/image";
    final static String SERVLET_ALIAS = "/example7_2";
    private HttpService http;

    public void start(BundleContext context) throws Exception {
        ServiceReference ref = context.getServiceReference(
            "org.osgi.service.http.HttpService");

```

```

http = (HttpService) context.getService(ref);
HttpContext hc = new HttpContext() {
    public String getMimeType(String name) {
        return null;
    }

    public boolean
    handleSecurity(HttpServletRequest req,
    HttpServletResponse resp) throws IOException
    {
        return true;
    }

    public URL getResource(String name) {
        URL url =
            this.getClass().getResource(name);
        return url;
    }
};

http.registerResources(IMAGE_ALIAS,
    "/servlet/image", hc);
http.registerServlet(SERVLET_ALIAS, new
    ExampleServlet(), null, hc);
}

public void stop(BundleContext context) throws Exception {
    if (http != null) {
        http.unregister(IMAGE_ALIAS);
        http.unregister(SERVLET_ALIAS);
    }
}
}
}

```

Code 7.16. File name: Activator.java

```

package servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Date;

class ExampleServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
        throws ServletException, IOException
    {
        String html =
            "<HTML>\n" +
            "<title>Example Servlet</title>\n" +
            "<body bgcolor=white>\n" +

```

```

        "<h1><font color=red>Time is
        Money!</font><h1>\n" +
        "<h3><font color=blue>This is a Sample of " +
        "Using HTTP Service.</font></h3>\n\n" +
        "<img src=\" + Activator.IMAGE_ALIAS +
        "/clock.gif\"><br>\n</body></HTML>\n";
    ServletOutputStream out = resp.getOutputStream();
    resp.setContentType("text/html");
    out.println(html);
    out.close();
}
}

```

Code 7.17. File name: ExampleServlet.java

```

Bundle-Activator: servlet.Activator
Import-Package: javax.servlet; specification-version=2.1,
    javax.servlet.http; specification-version=2.1,
    org.osgi.service.http

```

Code 7.18. File name: Manifest.txt

Step 2: Compile the two Java source files with the following command, where %JES_HOME% refers to the installation directory of JES 2.0.

```
javac -d . -classpath %classpath%;%JES_HOME%\lib\framework.jar; \
    %JES_HOME%\bundles\servlet.jar;%JES_HOME%\bundles\http.jar \
    *.java
```

Step 3: This example uses an image file. Select and put an image file (here we use `clock.gif` that comes along with JES 2.0) in the directory matching the source code 7.16 and 7.17: `.\servlet\image\clock.gif`.

Step 4: Generate a JAR file with the following command:

```
jar cmf Manifest.txt myServlet.jar servlet\*.class
    servlet\image\clock.gif
```

You can use a command “`jar tvf myServlet.jar`” to check the content of your JAR file, `myServlet.jar`, which should have the following content:

```
META-INF/
META-INF/MANIFEST.MF
servlet/Activator.class
servlet/Activator$1.class
servlet/ExampleServlet.class
servlet/image/clock.gif
```

Step 5: Start the JES 2.0 and install the bundle `myServlet.jar`. Since this bundle depends on `http.jar` and `servlet.jar`, make sure to start these two bundles before you install and start `myServlet.jar`.



FIGURE 7.6. Accessing a servlet registered with `HttpService`.

Step 6: Launch a Web browser with the URL `http://localhost:8080/example7_2`, you should see Figure 7.6 as a result.

Example 7.3

This example demonstrates the package connections among five OSGi components. The code is as follows:

```
package a;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
    public void start(BundleContext context) throws
        Exception{
        System.out.println("I am bundle a");
    }
    public void stop(BundleContext context) throws
        Exception{
    }
}
```

Bundle A's
Activator.java
And
Manifest file

```
Bundle-Activator: a.Activator
Import-package: d, b
```

<pre>package b; import org.osgi.framework.*; public class Activator implements BundleActivator { public void start(BundleContext context) throws Exception{ System.out.println("I am bundle b"); } public void stop(BundleContext context) throws Exception{ } }</pre>	<p>Bundle B's Activator.java And Manifest file</p>
<p>Bundle-Activator: b.Activator Import-package: c Export-Package: b</p>	
<pre>package c; import org.osgi.framework.*; public class Activator implements BundleActivator { public void start(BundleContext context) throws Exception{ System.out.println("I am bundle c"); } public void stop(BundleContext context) throws Exception{ } }</pre>	<p>Bundle C's Activator.java And Manifest file</p>
<p>Bundle-Activator: c.Activator Export-Package: c</p>	
<pre>package d; import org.osgi.framework.*; public class Activator implements BundleActivator { public void start(BundleContext context) throws Exception{ System.out.println("I am bundle d"); } public void stop(BundleContext context) throws Exception{ } }</pre>	<p>Bundle D's Activator.java And Manifest file</p>
<p>Bundle-Activator: d.Activator Import-Package: e, c Export-Package: d</p>	

```

package e;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
    public void start(BundleContext context) throws
        Exception{
        System.out.println("I am bundle e");
    }
    public void stop(BundleContext context) throws
        Exception{
    }
}

```

Bundle E's
Activator.java
And
Manifest file

```

Bundle-Activator: e.Activator
Export-Package: e

```

Code 7.19. Five simple bundles and their manifest files

According to their manifest files, these five components have the package connections illustrated in Figure 7.7.

Now, compile and package these codes into five JAR files and deploy them into JES framework. If you start bundle A first, what will happen? If you start B before C, what will happen?

Lab Practice 7.1

1. Download JES 2.0 from <http://wwws.sun.com/software/embeddedserver/buy/index.html> and install JES 2.0 on your computer, referencing the documentation of JES 2.0.
 - a. JES 2.0 was downloaded onto my computer but I failed to install it.
 - b. JES 2.0 was downloaded onto my computer and I have installed it successfully.
 - c. I have got the following problems: _____
 - d. I did not download or install it because it is already available on my computer.
2. How many different commands are provided by JES 2.0? _____ List five major commands that you have tried out so far.
 - a. _____
 - b. _____

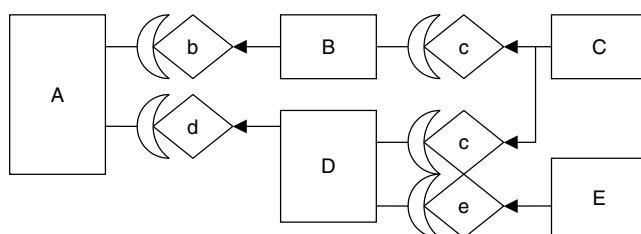


FIGURE 7.7. The package connections among five OSGi bundles.

- c. _____
- d. _____
- e. _____
3. Develop a Java program “`MyCommands.java`” similar to the sample code list 7.1. When started, the program displays the five JES commands you listed in Question 2. When stopped, the program displays “Bye XXX,” where “XXX” is your name.
4. Write down where is your cache directory. _____

Lab Practice 7.2

This assignment practices how to use framework console and the JES Management Panel to

- manage the bundle life cycle in the JES framework
 - obtain information about installed bundles
 - get and set system properties and OSGi environment variables
1. Launch the JES framework.
 2. Install and start the following three bundles: `log.jar`, `servlet.jar`, and `http.jar`.
 3. Make sure that only the three bundles in Step 2 are actively running in your framework. Now, issue the command “`exportedpackages`” to obtain a list of package dependencies. Using the graphical notations given in Figure 7.2 for OSGi package connections, draw a diagram illustrating the package dependencies among the three bundles.
 4. Make sure that only the three bundles in Step 2 are actively running in your framework. Now issue three commands “`manifest log`,” “`manifest http`,” and “`manifest servlet`” to obtain a list of services of these bundles. Alternatively, you can issue the command “`services`” and then filtered services command to collect services information: “`services (description =*Log*)`,” “`services (description =“Http”)`,” and so on. Following the graphic notations defined in Figure 7.4 for OSGi service connections, draw a diagram illustrating the service dependencies among the three bundles.
 5. Get OSGi Environment Properties:
 - a. Issue the command “`get org.osgi.framework.version`” to get the version of the framework.
 - b. Issue the command “`get org.osgi.framework.vendor`” to get the vendor of this framework implementation.
 - c. Issue the command “`get org.osgi.framework.language`” to get the language being used.
 - d. Issue the command “`get org.osgi.framework.os.name`” to get the name of the OS of the hosting computer.
 - e. Issue the command “`get org.osgi.framework.os.version`” to get the version number of the OS of the hosting computer.
 - f. Issue the command “`get org.osgi.framework.processor`” to get the name of the processor of the hosting computer.

6. Set OSGi Properties:
 - a. Issue the command “get com.sun.jes.service.http.hostname” to get the hostname of the computer running JES http service.
 - b. Issue the command “get com.sun.jes.service.http.port” to get the port number running JES http service.
 - c. Issue the command “set com.sun.jes.service.http.port=8080” to set the port number running JES http service.
7. The JES Management Panel (JESMP) provides a graphical interface to a framework instance. Make sure the three bundles in Step 2 have been installed and started. Then, issue the following commands: “start httpauth, tcatjspcruntime, httpusers, jesmp.” Now, check bundles with a command “bundles,” and you will find all the following seven bundles are in ACTIVE status: log.jar, servlet.jar, http.jar, httpauth.jar, tcatjspcruntime.jar, httpusers.jar, and jesmp.jar.
8. Start a web browser with the URL http://hostname:portnumber/admin in the location field. Fill in user name and password both as “admin.”
9. Print out your content of “View Log.”

Turn in: Your results of Steps 3, 4, and 9.

Lab Practice 7.3

Develop a grade service component in OSGi component infrastructure such that it stores student grades and displays the grades upon starting the bundle.

Step 1: Develop an interface, GradeService.java, publishing the grade service. For instance, you could provide a getGrade() method in the interface.

Step 2: Develop an implementation of the grade service interface. The implementation should store grades for a number of students and provide detail implementation of the getGrade() method. For instance, the method could be implemented in such a way that the student grade will be printed out upon request.

Step 3: Develop an activator class to test the grade service. Note that this activator class should register the grade service in its start() method and unregister it in its stop() method.

Step 4: Write a manifest file, and create a JAR file.

Step 5: Deploy your component into JES framework and start it.

Turn in: Your source files GradeService.java, GradeServiceImpl.java, Activator.java, and Manifest file.

7.6 SUMMARY

In the OSGi component infrastructure, the basic software components are called service bundles. A bundle usually consists of a service interface, an implementation class, and an activator class. The interface declares what services this component will provide, and the implementation class contains detail realization of how these services are implemented. The activator class conducts routine work for starting and stopping a service bundle, such as registers and unregisters its services with the framework. The

OSGi framework plays the role of a component home or component container. Its responsibilities include managing the life cycle of bundles, resolving interdependencies among bundles, maintaining a registry of services, and providing a communication platform for component collaborations. One of the most distinguishing features of OSGi component infrastructure is its support for dynamic evolution of a component-based system. OSGi components can be downloaded, installed, updated, stopped, uninstalled, and removed dynamically without terminating the overall system. OSGi components are highly independent of each other in that code within one bundle cannot refer to classes inside another component. This feature protects classes in one component from interfering with those in another component. However, components in OSGi model do not completely isolated from each other. They are hooked to the OSGi framework and they cooperate through the OSGi connection model.

The OSGi connection model allows cooperation among service bundles as well as creation of large applications using bundles as the building blocks. Thus, a high-level bundle could rely on lower-level bundles to do those common and routine works. There are two kinds of component connections in OSGi: package connections and service connections. One bundle can export its packages to make the classes available for other bundles to use. On the other hand, a bundle can request other packages by importing them. Both import and export packages are done by declaring the packages using headers in their manifest files. Package connections represent static dependencies among components because these connections are specified at design time by the component developers. The service connections, however, represent dynamic dependencies among components as they are resolved at runtime with the help of OSGi framework. When one bundle registers its services with the framework, it publishes its services so that other bundles can obtain and consume the services. The dynamic characteristics of OSGi service bundles create unique challenges for handling service connections. What would happen if one client component is actively using the services provided by a server component and the server bundle decides to withdraw?

The OSGi framework solves the dynamic service dependency by event handling. Within the framework, events are broadcast when bundles go through their life cycle states and when services are published or withdrawn. A server component withdraws its service by unregistering its services with the framework. The unregistering action is not complete until all the client components have finished their actions and have done proper cleanup to release the service connections. It is important to notice these dynamic features inherent in OSGi component infrastructure when you build your service bundles.

The deployment model in OSGi component infrastructure relies on the framework implementation for the OSGi specification. Under JES 2.0, an implementation of OSGi 1.0 provided by Sun Microsystems Inc., the deployment of a component is done by supplying the framework with the URL of the bundle's JAR file, which may reside on the local computer or on a remote server. The framework will assign a unique ID for each service bundle, resolve its package dependency and service dependency, and provide other necessary resources for it to be functioning correctly. When a bundle is to be stopped, the framework calls the bundle activator's `stop()` method and unregisters the services provided by this bundle.

The OSGi component infrastructure enforces the separation of interface and implementation. Thus, any change made to the implementation is transparent to the client of the component. This will encourage software reuse and improve flexibility as well as

reduce maintenance cost. The OSGi specification focuses on delivering multiple services through a gateway in home, office, or vehicle environment. However, the OSGi component infrastructure merits further research and study from a broader software engineering perspective [Koza 1999; Parrish 1999].

7.7 SELF-REVIEW QUESTIONS

1. The OSGi in this chapter stands for
 - a. Operating System General initiative
 - b. Operating System Generic interface
 - c. Open Service Generation and interconnection
 - d. Open Service Gateway initiative
2. Which of the following is the component in OSGi component infrastructure?
 - a. Imported package
 - b. Exported package
 - c. Service bundle
 - d. Manifest file
3. If a service bundle has an activator class, what methods have to be implemented in this class?
 - a. Start and stop
 - b. Run and sleep
 - c. Constructor and destructor
 - d. Register and unregister
4. When you compile your source code in order to generate an OSGi component, which JAR file should be included in your CLASSPATH?
 - a. %JAVA_HOME%/lib/tools.jar.
 - b. %JES_HOME%/lib/framework.jar.
 - c. %JAVA_HOME%/lib/dt.jar.
 - d. %JES_HOME%/bundles/log.jar.
5. Suppose you have created a correct manifest file and you have two class files in the same working directory to be included in the JAR file. Which of the following is the correct command to create the myBundle.jar file?
 - a. jar cmf myBundle.jar Manifest myclass1.class myclass2.class
 - b. jar cfm myBundle.jar Manifest myclass1.class;myclass2.class
 - c. jar cmf Manifest myBundle.jar myclass1.class/myclass2.class
 - d. jar cmf myBundle.jar Manifest myclass1.class myclass2.class
6. What is the OSGi framework?
 - a. It is the service bundle that an OSGi component developer must develop and deploy first.
 - b. It is the Java virtual machine for OSGi services to function correctly at runtime.

- c. It serves as the common environment for hosting a set of OSGi components called bundles.
 - d. It is an interface between gateway hardware and gateway software such as operating systems.
7. If a manifest file for a nonlibrary OSGi component has only one line, what header will it specify?
- a. Export-Package
 - b. Import-Package
 - c. Bundle-Version
 - d. Bundle-Activator
8. In the OSGi component connection model, if a bundle *imports* a package, it indicates that:
- a. The bundle informs the framework to make the classes in the exported package available for any other bundles that need to use them.
 - b. The bundle is looking for a peer bundle that imports the same package.
 - c. The bundle needs to use the specified classes from the imported package in order to run.
 - d. The bundle is communicating to a peer bundle directly for a connection.
9. In the OSGi component connection model, if a bundle *exports* a package, it indicates that:
- a. The bundle informs the framework to make the classes in the exported package available for any other bundles that need to use them.
 - b. The bundle is looking for a peer bundle that imports the same package.
 - c. The bundle needs to use the specified classes from the imported package in order to run.
 - d. The bundle is communicating to a peer bundle directly for a connection.
10. In OSGi component connection model, which of the following is correct?
- a. The bundle package connection is independent of service connection.
 - b. The bundle package connection is a prerequisite for service connection.
 - c. The bundle service connection is a prerequisite for package connection.
 - d. The bundle service connection is complete as soon as package connection is resolved.
11. In the OSGi component connection model, which of the following is correct?
- a. The bundle package connection is dynamic.
 - b. The bundle service connection is static.
 - c. The bundle service connection is dynamic.
 - d. The package connection and service connection both are dynamic.
 - e. The package connection and service connection both are static.
12. Component-oriented programming with OSGi component technology requires more discipline on the part of programmers because:
- a. The programmers have to learn Java.

- b. The programmers can always count on other components to provide necessary services.
- c. Once the services provided by other components have registered, they will be available all the time.
- d. The services a component depends on may not always be there.

Keys to Self-Review Questions

1. d 2. c 3. a 4. b 5. a 6. c 7. d 8. c 9. a 10. b 11. c 12. d

7.8 EXERCISES

1. What is the component model in OSGi component infrastructure?
2. What is the connection model in OSGi component infrastructure?
3. What is the deployment model in OSGi component infrastructure?
4. What is the difference between STARTING and ACTIVE states in a bundle life cycle?
5. Under what situations could a bundle enter RESOLVED state?
6. When the `stop()` method in a bundle's activator class is called, what tasks will be performed by the OSGi framework?
7. Develop an OSGi component providing billing services for a home user. Typical functions of such billing services include:
 - a. On-line billing service for home internet connection.
 - b. Automatic billing service for local and long-distance phone bills.
 - c. Automatic billing service for cable or satellite TV services.
 - d. Automatic billing service for water, electricity, and gas companies, assuming that smart meters have been installed at home to automatically report meter readings to a home gateway.
8. Develop an OSGi component to check your local news.
9. Develop an OSGi component to provide fax services for your home business.
10. Develop an OSGi component to control your home light system, including gate lights, garage lights, back yard lights, bedroom lights, and so on.
11. Develop an OSGi component to monitor and control your home temperature.
12. Develop an OSGi component to monitor and control your lawn sprinkler.
13. Modify the grade service in Lab 7.3 such that when started, the service will display the class average in addition to individual grade.
14. Develop a service bundle based on Lab 7.3 to make the service interactive. When the bundle starts, it will first ask the user to input her/his name, then the service searches its database and displays the corresponding grade if there is a match between the user name and one record in its own database.

15. Write a short research paper discussing the commonality and differences between OOP and COP with the OSGi component infrastructure.
16. Write a term paper comparing the OSGi component infrastructure with other component infrastructures such as JavaBeans, EJB, CCM,. NET, Web Services, and so on.

REFERENCES

- [Brown 1998] Brown, A. W. and Wallnau, K. C. "The current state of CBSE," *IEEE Software*, Sept/Oct: 1998, 37–46.
- [Chen 2002] Chen, K. and Gong, Li. *Programming Open Service Gateways with Java Embedded Server Technology*, Addison-Wesley, New York, 2002.
- [Koza 1999] Kozaczynski, W. "Composite nature of component," *1999 International Workshop on Component-Based Software Engineering*, <http://www.sei.cmu.edu/cbs/icse99/papers>, 1999.
- [Oscar 2003] Hall, R. S. <http://sourceforge.net/projects/oscar-osgi/>, accessed in May 2003.
- [OSGi 2003] OSGi Alliance Web site: <http://www.osgi.org>, visited in May 2003.
- [Parrish 1999] Parrish, A., Dixon, B., and Hale, D. "Component based software engineering: a broad based model is needed," *1999 International Workshop on Component-Based Software Engineering*, <http://www.sei.cmu.edu/cbs/icse99/papers>, 1999.
- [ProSyst 2003] ProSyst Software AG. <http://www.prosyst.com/>, accessed in May 2003.
- [Sun 2003] Sun Microsystems, Inc. <http://www.sun.com/software/embeddedserver/>, accessed in May 2003.

8

WEB SERVICES COMPONENTS

Objectives of This Chapter

- Introduce the Web services framework
- Introduce concepts of Web services components
- Discuss compositions of Web services components
- Discuss types of Web services components and their deployments
- Discuss the interoperability of Web services components
- Distinguish between synchronous and asynchronous Web services invocations
- Provide step-by-step tutorials on building, deploying, and using Web services components

8.1 WEB SERVICES FRAMEWORK

8.1.1 Overview of the Web Services Framework

Web service is a new paradigm to deliver application services on Web and enables a programmable Web, not just an interactive Web. Web service is the third generation in the Web evolution after static HTML and interactive Web development such as PERL, ASP, JSP, and others. Web services are typical black box-reusable building block components in the distributed computing. Although there are many other distributed component frameworks available in the industry such as CORBA, MS DCOM, .NET, and EJB, only Web service can provide cross-platform, cross-programming language, cross-proprietary restriction, and internet firewall-friendly solutions to the interoperable distributed computing. All other distributed technologies face

some painful problems such as difficulties of message exchanging and remote object method invocation between different platform components, for example; we need to bridge the gaps between them. Furthermore, each of these technologies needs to have some special protocol layers to be installed such as ORB/IOP for CORBA, which makes the interoperability very difficult in the distributed computing. Most of these distributed component technologies have a hard time to deal with the firewall security. Web service works on a protocol stack consisting of SOAP/XML/HTTP/TCP/IP, which makes Web services widely acceptable as long as clients of Web services have supports from these protocols. Web service revolutionizes the distributed computing and signals a new era of lightweight distributed application development in the advantage of loosely coupled features of Web services components. The deployment of components in other technologies is not very easy. It is very easy to deploy, publish, discover, and invoke a Web service on Internet. Web service is a solution to all the above problems. There is a strong shift in e-Commerce, B2B, Enterprise Application Integration (EAI) from using proprietary technologies to using Web services to develop new applications on-line or to wrap existing application with Web services technologies. There is no precise definition on Web services by a single statement. We can simply say that Web service is a self-descriptive on-line distributed component, which exposes its services and functionality via its interface on-line and can be published, located, and invoked programmatically over the Internet. A key point of the importance of Web services is its ability to support programmatic endpoint for any application to get services provided by any Web service on-line.

One philosophy behind Web services is to shift distributed software development from programming to composition, from coding ground up to building new application from existing components either by purchasing or by reusing the existing components. This is exactly the advantage of Component-Oriented Programming (COP) and Component Based Software Development (CBSD). So the main goal of Web services is not only to provide services on Web but also to provide a mechanism to share its services as a building block to be part of other Web services or application programs via its programmatic Web service endpoints. Web service has shown its potentials to change the ways of the enterprise business modeling in the distributed system interactions. A large enterprise system can be divided into many relatively independent Web service components, which in turn get services from other Web services components deployed on the Internet. EAI or B2B can be constructed in a hierarchy of Web services components. Some of these Web services components can be developed from scratch and others can be constructed on the basis of the existing Web services compositions or by converting some existing distributed components to be Web service available such as EJB components.

There are already thousands of Web services available on-line. We can predict that more and more Web services will be there and more and more Web applications will take advantages of Web services. Many Web services development tools are there such as Apache Tomcat next generation Web service – AXIS, Microsoft .NET Web service studio based on IIS server, IBM Web Sphere Web service, BEA Web Logic Workshop, Java Web Service Development Pack (JWSDP), Mind Electric GLUE, and others. All of them provide development tool kits for Web services developments and deployments. We will discuss two Web services servers, Apache Tomcat AXIS and Microsoft .NET IIS Web services in detail in this chapter.

Let us summarize the characteristics and features of the Web services technology as the following:

- It increases the portability and interoperability of distributed computing.
- It increases the reusability and scalability of distributed components.
- It reduces the complexity of component composition and deployment.
- It reduces the cost and time-in-market for distributed component software developments.
- It significantly enhances the B2B and Electronic Data Interchange (EDI).
- It simplifies the distributed system administration.
- It is very easy to wrap an existing proprietary application to be Web services compatible.
- It operates on an open standard protocol stacks.
- It is a third-generation Internet solution for distributed computing.

The Unified Communication Technical Project in UC Berkley is a good example of Web services applications to unify all e-mail services, voice-mail services, PDA services, fax services, and other electronic or computer-based services by Web services technologies. Web services also face some challenges such as QoS in terms of response time, performance, and Service Level Agreement (SLA) problems.

We will explore the basic concepts of XML and SOAP in next few sections since they are parts of the foundation protocol stacks of Web services.

8.1.2 XML Basics

eXtensible Markup Language (XML) is a super set of HTML. A user can define his/her own tags in XML. Any XML document must follow its metadata such as Document Type Definition (DTD) or Schema, which specifies complex data types, elements, its attributes, and its subelements, and so on. An XML document can be used to represent and transfer structured data in the hierarchy of element tags. An XML documents with its schema or DTD can be recognized by any programs or software easily as long as XML API is supported. An XML document is a universal format document type used for data exchange and data storage. Almost all commercial databases support XML now. Oracle 9i supports XML format database, which is one step beyond the import and export of XML documents. XML is also widely used for deployment descriptors and configuration specifications.

Let us look at a simple XML document about *student* GPA record.

```
<?xml version = "1.0"?>
<students>
    <student id=1234>
        <name>John Smith</name>
        <gpa>3.5</gpa>
    </student>
    <student id=2345>
        <name>Scott Tiger</name>
        <gpa>4.0</gpa>
    </student>
</students>
```

This XML document shows two student records. Each record consists of the student's name and his/her grade point average (gpa). The tag **students** is a top-level root element, which has a number of subelements called **student**. Each **student** element consists of two subelements called **name** and **gpa**. The identifier **id** in **student** tag is an attribute of element **student**. The tag “?” is a Processing Instruction (PI) to inform the XML parser that this XML document must conform to XML v. 1.0. Every XML document must have its metadata just as a data record in a data table of any database must satisfy the definition of the table, which is called *schema* in database. There may be many different formats of XML schema such as internal DTD within XML file, a separate external DTD, or a schema, since the DTD itself does not use the XML syntax and is not very flexible. Schema is getting more and more popular now and is replacing DTD because schema itself is in XML format. We will focus on Schema describing the XML structure. An XML Schema Document (XSD) is a metadata of an XML document. XSD specifies the syntax, structure, and constraints in a corresponding XML, including data type or complex data type of elements, attributes of elements, and so on. The following Schema describes the structure of the above **students** XML document. A complex type can be specified by a **complexType** tag as follows.

```
<element name = "student">
    <complexType>
        <element name = "name" type = "xsd: string" />
        <element name = "gpa" type = "xsd: float" />
    </complexType>
</element>
```

Let us examine a complete XSD Schema sample.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.mybooks.org"
    xmlns="http://www.mybooks.org">
    <xsd:element name = "students">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="student"
                    minOccurs="1"
                    maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="student">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="name" />
                <xsd:element ref="gpa" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="gpa" type="xsd:float" />
</xsd:schema>
```

The sequence tag in XSD Schema specifies the order and the occurrence of subelements in their parent element. In this example, name and gpa elements are the subelements of the student element, which in turn is an element of the students element. Of course there may be many students in a students set. The XMLNS plays the same role as a package in Java and namespace in C++ to prevent naming collisions. An element name can be qualified with a namespace prefix with a colon symbol “:”. The idea of combining a namespace URI with a local name is to make any identifier name in XML universally unique.

For example, book in `<xyz:book xmlns:xyz =http://www.amazon.com/books/>` is an element name and xyz is the prefix, which is defined in xmlns namespace construct with the unique URI address as `http://www.amazon.com/books`.

8.1.3 Simple Object Access Protocol (SOAP) Basics

SOAP is an XML-based message exchange protocol specified by the W3C specification. Its new version SOAP 1.2 was recommended by W3C in 2003. SOAP is also a lightweight protocol working in a distributed heterogeneous environment. Some people simply say that $\text{SOAP} = \text{HTTP} + \text{XML}$ because a SOAP message is an XML document but it conforms to a specific XML Schema. SOAP is used to specify the format of a request and response in the Web services computing to get and send messages via HTTP port by HTTP POST method.

Every SOAP message has a required envelope and a message body. A SOAP envelope identifies an XML document as a SOAP message. A SOAP envelope element must have one subelement, which is the Body element. A Body element of a SOAP message can be data itself, a name of a method to be invoked, arguments of the method to be invoked, a SOAP request message, or return results from a SOAP response message. A SOAP envelope may also have a header subelement, but it is optional. A header element just gives more metadata about the message.

Here is an example of a SOAP request message that asks a Web service to convert temperature zero in Celsius degree to Fahrenheit degree. The name of the invoked method is “toFahrenheit.”

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-
        ENV="http://schemas.xmlsoa.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-
        ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
    <toFahrenheit>
        <celsius xsi:type="xsd:string">0</celsius>
    </toFahrenheit>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the Body element of this SOAP request, an XSD string type data “0” is passed as an argument to the remote method (operation) `toFahrenheit` of this Web services.

The following SOAP message is a SOAP response message of the converted temperature of 32 in Fahrenheit degree:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <toFahrenheitResponse
            SOAP-ENV:encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/">
            <toFahrenheitResult xsi:type=
                "xsd:string">32</toFahrenheitResult>
        </toFahrenheitResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

8.1.4 Web Services Architecture

There are three important components in Web services architecture. Figure 8.1 shows the Web service requester (simplified client) on the left, the Web service provider on the right, and the Web service registry on the top.

A Web services provider must publish/register its services with a Universal Description, Discovery, and Integration (UDDI) registry so that it can be accessed by any Web services requester globally. It just looks like a phonebook, where all businesses register their phones there for customers to look up services. A customer must look up the phonebook either on-line or by phonebook unless a customer knows the phone number before. A Web services can also be reached without any assistance from UDDI if the Web services client knows the contact information such as Web services's URL, name of method, argument signature of the method including types

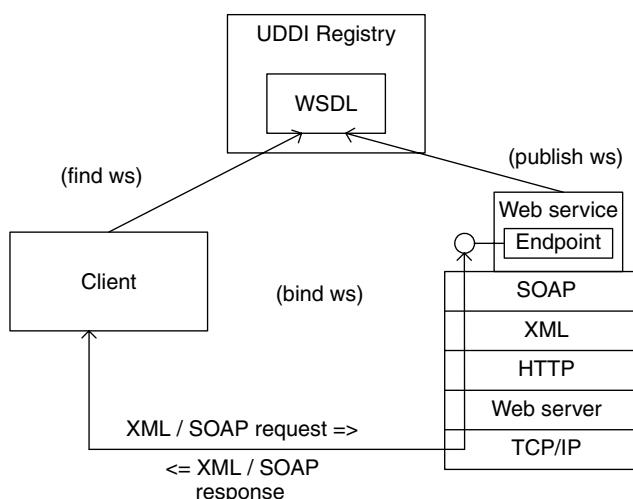


FIGURE 8.1. Web services architecture.

of return value. There are a number of UDDI registries available on-line such as uddi.microsoft.com, uddi.ariba.com, and www-3.ibm.com/service/uddi. The Web services provider can register their Web services at these UDDI registries and Web services requester can look up a specific Web service by its provider name (business name), category info, Web service name, or even by their keys if the requester knows them. A Web services provider actually registers a WSDL (Web services Description Language) interface at UDDI registry, which is a contract interface of the Web services to be used by its client. Both UDDI query requests and UDDI query results are in the SOAP formats [Deitel 2003a, 2003b; McGovern 2003].

In the following section, we will give more detail discussions on WSDL and UDDI.

8.1.4.1 WSDL A Web services WSDL interface is an XML file, describing what functionality this Web service provides, where this Web service resides, and how to access Web service and invoke the remote method provided by this Web service. WSDL shows an abstract view or interface definition that can be implemented by many concrete Web services implementations in different platforms and in different languages. We can divide a WSDL Web services definition into two parts: Web services reusable interface specification and Web services implementation. There may be four elements in first part: The **Types** elements for complex data type and user-defined type; **Message** elements for message description, which is used in **portType** element; **PortType** element describing an abstract set of reusable exposed operations by one or many Web service endpoints(**port**); **binding** element for message format specification and concrete protocol binding specification such as SOAP, HTTP, and MIME. There are two major elements in WSDL implementation part: **Port** element, which is a single Web services endpoint defined on the basis of a binding defined in binding element and an URL access location; **Service** element, which is a parent element of port element and a collection of related Web services endpoints or ports. WSDL XML document basically defines Web services as a collection of endpoints or ports.

The structure of a WSDL looks like the following templates:

```
<definition ...>
  <types ... >
  <message ...>
  <portType ... >
  <binding ... >

  <service ... >
    <port ... >
  </service>
</definition>
```

The following list is a WSDL definition part of `Convert.wsdl`, which gives a clear description of a Web services definition. A WSDL definition element has a number of subelements such as **message** element, **portType** element, **binding** element, **port** element, and **service** element.

```
<wsdl:definitions . . .
  <wsdl:message name="toFahrenheitResponse">
    <wsdl:part name="return" type="SOAP-ENC:string"/>
```

```

</wsdl:message>

<wsdl:message name="toFahrenheitRequest">
    <wsdl:part name="in0" type="SOAP-ENC:string" />
</wsdl:message>

<wsdl:portType name="Convert">

    <wsdl:operation name="toFahrenheit" parameterOrder="in0">
        <wsdl:input message="intf:toFahrenheitRequest" />
        <wsdl:output message="intf:toFahrenheitResponse" />
    </wsdl:operation>

</wsdl:portType>

<wsdl:binding name="ConvertSoapBinding" type="intf:Convert">

    <wsdlsoap:binding style="rpc"
                      transport="http://schemas.xmlsoap.org/soap/http" />

    <wsdl:operation name="toFahrenheit">

        <wsdlsoap:operation soapAction="" />

        <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:myDirectory" use="encoded" />

        </wsdl:input>

        <wsdl:output>

            <wsdlsoap:body
encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:myDirectory" use="encoded" />

        </wsdl:output>

    </wsdl:operation>

</wsdl:binding>

<wsdl:service name="ConvertService">

    <wsdl:port binding="intf:ConvertSoapBinding" name="Convert">

        <wsdlsoap:address
location="http://localhost:8080/axis/services/Convert" />

    </wsdl:port>

</wsdl:service>

</wsdl:definitions>

```

The name of the Web services “ConvertService” is specified in a service element, which is listed at the end of the above WSDL file. The port element of the service element specifies an URL address of the Web service and access point of this Web service for a unique binding. The binding “ConvertSoapBinding” is defined in a binding element, which is referenced in binding attribute of port subelement of service element in WSDL definition element. There may be multiple ports in one Web services.

There are two message elements at the beginning of definition element. One describes the argument type of Web services’s remote method and the other is the return type of the same method. The part subelement specifies the name and data type of the message exchanged.

The portType element describes an operation provided by the Web services. An operation is a method that a client must know about the input parameters and return data type. The two messages defined for Web services request and response are specified in the operation element. The operation element describes that the name of the method in Web service is `toFahrenheit`. A portType is a collection of operations.

The binding elements inside definition element specify how a client and Web services should exchange message with each other. The binding subelement of the binding element specifies SOAP as the protocol, and input request and output response must be in a SOAP format. It also tells that this is a request/response two-ways operation by the attribute style “`rpc`”. The other style option may be “`document`”. There are four type of operations in terms of patterns of inputs and outputs: `input only`, `output only`, `input/output`, `output/input`. These will be discussed in detail in the Web services connection section.

8.1.4.2 UDDI UDDI is a technical specification for building a distributed directory for business and Web services, which enables business companies to publish and find Web services. How to publish a Web service globally and how to discover a desired Web service to reuse it as part of new Web service or as part of client application is a challenge for Web services to be widely used. If a client of a Web service knows in advance about the location of the Web services and the way to invoke the operations provided by the Web services, there will not be a need for Web services registry. UDDI acts like a naming service in the distributed computing, or phone directory for phone service, or Google search engine for Internet service. UDDI consists of an XML schema that defines UDDI’s four core data structures – business, service, binding and tModel programmatic interface, and a set of APIs that provide publishing and inquiry operations on those structures. UDDI was developed by IBM, Microsoft, and Ariba in 2000, and is now under the stewardship of the UDDI community (www.uddi.org), which has more than 200 member companies. UDDI is currently at Release Version 3.0.

In Figure 8.1, we see that a Web services requester searches the service registry and finds the desired service description. Through the information the requester finds in the registry, the requester connects to the Web services provider and invokes the service. UDDI is a group of specifications that lets Web services providers publish information about their Web services and lets Web service requesters search that information to find a Web service and run it.

A Web services listing is created using WSDL and then sent to a UDDI registry, which is mapped to a UDDI XML format document. A listing is composed of three elements. At the highest level, there are *White Pages*, which contain basic information about the business, including business name, descriptions, contact info (name, address,

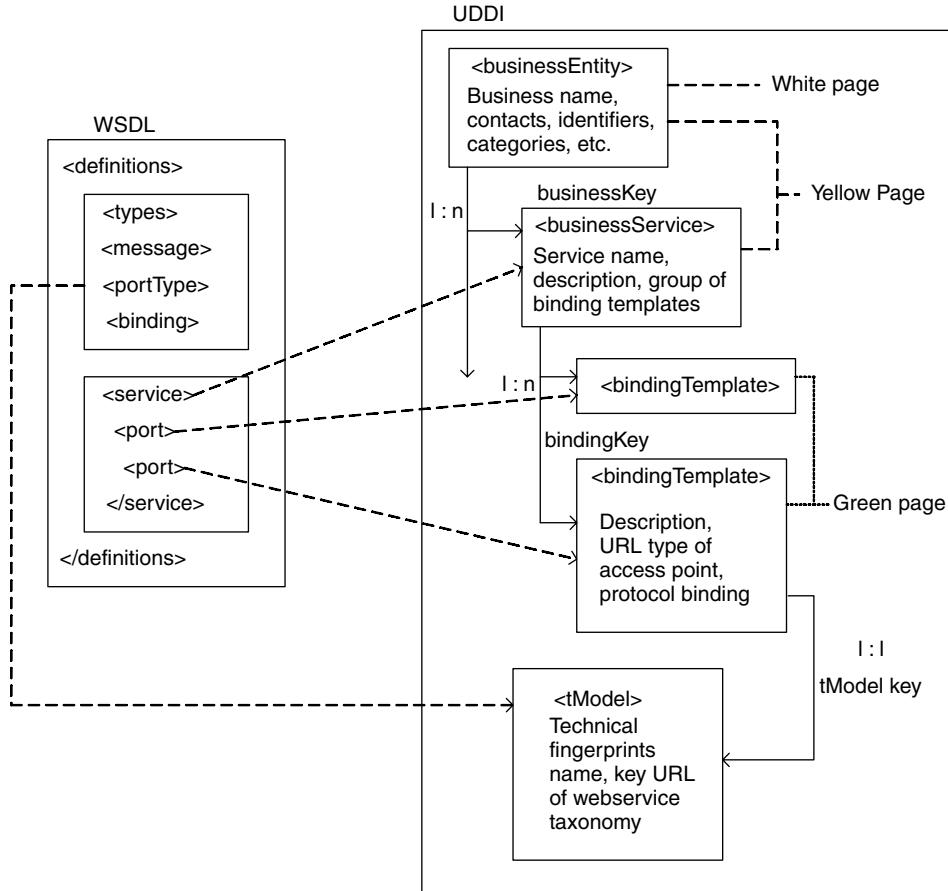


FIGURE 8.2. Web service WSDL to UDDI mapping.

phone, fax, Web site, etc.), and business identifier that a business may be known by. Next are *Yellow Pages*, which organize services by industry codes, service type/business categories in product/services or geographical location taxonomy. Finally, there are *Green Pages*, which specify how to bind to a service provider. It includes the technical information (such as interfaces and URL locations) about how to find and execute a Web service. An application requesting a service will use WSDL to programmatically interact with the Green Pages section of that service's listing.

In short, UDDI is organized in a three-level nested structure model with business information, service information, and binding information.

In Figure 8.2, we can see a very clear picture of UDDI and how a WSDL is mapped onto UDDI.

First of all, let us focus on the data structure and then we will discuss how to publish a Web service, how to find, and how to get a Web service by UDDI's API.

```

<businessDetail>
  <businessEntity businessKey="[BSK1]">
    Names, Descriptions, Contacts, ...
  
```

```

<businessServices>
  <businessService serviceKey="[SK1]" businessKey="[BSK1]">
    Names, Descriptions, ..
    <bindingTemplates>
      <bindingTemplate bindingKey="[BDK1]" serviceKey="[SK1]">
        Descriptions ..
        <accessPoint
          URLType="http">http://www...</accessPoint>
        <tModelInstanceDetails>
          <tModelInstanceInfo>
            <tModelKey>[tMK1]</tModelKey>
            Descriptions, ..
            <instanceDetails>
              Descriptions, Overview URLs, ..
            </instanceDetails>
          </tModelInstanceInfo>
        </tModelInstanceDetails>
      </bindingTemplate>
      ...
    </bindingTemplates>
    <categoryBag>...<tModelKey>[tMK1]</tModelKey>...
    </categoryBag>
  </businessService>
  ...
</businessServices>
<identifierBag>...<tModelKey>[tMK1]</tModelKey>...
</identifierBag>
<categoryBag>...<tModelKey>[tMK2]</tModelKey>...</categoryBag>
</businessEntity>
...
</businessDetail>

```

This business detail information can be retrieved by a SOAP request if a business key is known in advance. The following SOAP XML shows the SOAP request for the above information:

```

<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns='http://schemas.xmlsoap.org/soap/envelope/'>
  <Body>
    <get_businessDetail xmlns="urn:uddi-org:api">
      <name>[BSK1]</name>
    </find_business>
  </Body>
</Envelope>

```

We can also get the following tModelDetail information about a specific Web service by a SOAP request against UDDI if we know tModelKey in advance.

```

<tModelDetail ... >
  <tModel tModelKey = "[tMK1]" operator = " .. " .. >

```

```

<name>myMethod .. </name>
<description> .. </description>
<overviewDoc>
    <description> .. </description>
    <overviewURL>http:// ... myWsdl.wsdl</overviewURL>
</overviewDoc>
<categorybag>
    <keyedreference tModelKey =“[tMK3]“
                    keyName=“uddi-org-types“
                    keyValue=“wsdlSpec“/>
</categorybag>
</tModel>

...
</tModelDetail>
```

The [XXX] in the above UDDI examples are the keys for `business`, `webservice`, and `tModels`. The `myWsdl` is the ultimate endpoint of this Web services. There are many ways to get this endpoint; either by drilling down step by step from a known business name, service name, or by a key id such as business key, Web services key, key of `BindingTemplate`, or `tModelkey`.

UDDI specifications specify a programmer's API to allow programmatic access to UDDI registry information. This API is divided into two logical parts: publisher API and inquiry API. The publisher API is used to publish and update the information stored in a UDDI registry. The syntax of publisher UDDI API elements can be `save_XX` or `delete_XX`, where XX can be business, service, binding, or `tModel`. The Inquiry API is used to find and get Web services information in a UDDI registry. The syntax of a UDDI inquiry API element can be `find_XX`, which can be used to search for a broad overview of registration data based on a variety of criteria or can also be `get_XXDetail` direct call if an actual key is known in advance. XX can be business, service, binding, or `tModel`, same as in publisher API.

Here is a sample SOAP UDDI query to find “SUN” business registered in a UDDI registry by `find_business` SOAP request message.

```

<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns='http://schemas.xmlsoap.org/soap/envelope/'>
<Body>
<find_business generic="1.0" xmlns="urn:uddi-org:api">
    <findQualifiers></findQualifiers>
    <name>SUN</name>
</find_business>
</Body>
</Envelope>
```

The response that comes back from UDDI may look like the following.

```

<soap:Envelope ...>
<soap:Body>
<businessList generic="1.0" operator="...">
```

```

    truncated="false" xmlns="urn:uddi-org:api">
<businessInfos>
    <businessInfo businessKey="[BSK1]">
        <name>SUN Microsystems</name>
        <description xml:lang="en"> ... </description>
        <serviceInfos>
            <serviceInfo businessKey="[BSK1]"
                         serviceKey="[SVCK1]">
                <name>Products..</name>
            </serviceInfo>
            <serviceInfo businessKey="[BSK1]"
                         serviceKey="[SVCK2]">
                <name>Developer..</name>
            </serviceInfo>
        </serviceInfos>
    </businessInfo>
</businessInfos>
</businessList>
</soap:Bbody>
</soap:Envelope>

```

It turns out that SUN Microsystem's business with `businessKey BSK1` encapsulates two services with `serviceKey SVCK1` and `SVCK2`, which in turn encapsulate `bindingTemplate`. The `bindingTemplate` is not shown in this list. If we drill down further by an UDDI query for details by `get_businessDetail` with its business key, it will return all details including a reference pointing to a `tModel` of technical model of Web services access detail information.

```

<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns='http://schemas.xmlsoap.org/soap/envelope/'>
<Body>
<get_businessDetail generic="1.0" xmlns="urn:uddi-org:api">
    <findQualifiers></findQualifiers>
    <businessKey>[BSK1]</businessKey>
</get_businessDetail>
</Body>
</Envelope>

```

The response SOAP message is shown below.

```

<soap:Envelope xmlns:soap = " ... " ... xmlns:xsd = " ... "
<soap:Body>
    <businessDetail .. >
        <businessEntity businessKey="[BSK1]" ... >
            <discoveryURLs>
                <discoveryURL ..
                    businessKey=[BSK1]>[URL1]</discoveryURL>
            <name>Sun ... </name>
            <description ... >... </description>
            <contact>... </contact>
            <businessServices>

```

```

<businessService serviceKey="[SVK1]" businessKey="[BSK1]">
    <name>...</name>
    <description>...</description>
    <bindingTemplates>
        <bindingTemplate serviceKey="[SVK1]"
                         bindingKey="[BDK1]">
            <accessPoint URLType="[URL2]"></accessPoint>
            <tModelInstanceInfo tModelKey="[TMK1]" />
        </bindingTemplate>
    </bindingTemplates>
    </businessService>
</businessServices>
<categoryBag>
    <keyedreference tModelKey="[TMK2]"
                   keyName="[K1]" keyValue="[V1]" />
    ...
</categoryBag>
</businessEntity>
</businessDetail>
</soap:Body>
</soap:Envelope>

```

Here we can find that a `businessEntity` may encapsulate many `businessServices`, each `businessService` may encapsulate many Web services, and each Web service can encapsulate and may bind many bindings. Each binding references to its unique `tModel` that provides all technical detail information to get this Web service. The following shows an example of `tModel` SOAP message referenced by a `tModelKey` TMK1.

```

<soap:Envelope ...>
    <soap:Body>
        <tModelDetail ...>
            <tModel tModelKey="[TMK1]" ... >
                <name> ... </name>
                <description> ... /description>
                <overViewDoc>
                    <description> ... </description>
                    <overviewURL>http://.../XYZ.wsdl</overviewURL>
                <overViewDoc>
                <categorybag>
                    <keyedReference tModelKey=... keyname=...
                        keyValue=.../>
                </categoryBag>
            </tmodel>
        </tModelDetail>
    </soap:Body>
</soap:Envelope>

```

UDDI plays a registry role for business to publish its Web services and for Web services application to find a needed Web services for their applications.

8.2 COMPONENT MODEL OF WEB SERVICES

8.2.1 Interface and Implementation of Web Service Component

As we have discussed so far, Web services is a typical software component available on-line. There is also a Web services interface just like other components but Web services interface is specified in a WSDL XML format file instead of an IDL file in CORBA. We can have a single wsdl interface specification including all necessary information for a Web services client to access it, as shown below:

`myService.wsdl:`

```
<definition ...>
    <types ... >
    <message ...>
    <portType ... >
    <binding ... >

    <service ... >
        <port ... >
    </service>
</definition>
```

The first four elements constitute a reusable Web services interface definition. The type element describes the XSD type and user-defined types that may be used in message element. The message element describes the request and response message specifications used in a portType element. The portType element specifies a collection of operations (methods) exposed by Web services. PortType plays a very similar role of class in Java. The binding element binds and maps operations defined in portType element to a transport protocol. The service element plays a role of implementation of the above Web services definition by specifying the binding defined and physical endpoint address of the Web services where it resides.

We can also have two or multiple WSDL specifications working together that one WSDL can import definitions from another WSDL. For example, we can separate myService.wsdl into two WSDL specifications, service implementation WSDL myServiceImpl.wsdl and service interface WSDL myServiceIntf.wsdl. The service interface gives a reusable definition of a Web service and corresponds to a tModel in UDDI. The service implementation is an implementation of the service. A WSDL XML specification can be generated from an implementation Java source file or even from a Java interface source file by Web services development tools. For example, java2wsdl utility command line can be used to generate or make an internet URL request a [Web services name].jws?wsdl to get and save it in a file.

Service interface document myServiceIntf.wsdl:

```
<definition ...>
    <types ... >
    <message ...>
    <portType ... >
    <binding ... >
</definition>
```

Service implementation document `myServiceImpl.wsdl`:

```
<definition ...>
    <import namespace= ... location="... /myServiceIntf.wsdl">
    <service ... >
        <port ... >
    </service>
</definition>
```

In `myServiceImpl.wsdl`, the import element has a location attribute, which references to the service interface document and a namespace attribute, which matches the `targetNamespace` in the service interface document. The service element specifies the actual location (URL) of Web services on the server, and port element is the endpoint of Web service, specifying the address for particular binding.

A WSDL specification can be generated from an existing Web services implementation such as a Java source file or from a Java interface file.

In order to understand Web services better, we choose the popular third-generation Web services – Apache eXtensible Interaction System (AXIS) to explain it in detail. AXIS is an extension or replacement of Apache SOAP Web service engine. There are many other Web service runtime environment engines available such as .NET Web service engine for which we will show some practice examples in the lab section.

Basically, there are two different ways to develop a Web service, depending on the existence of service implementation:

- *Top-Down Design* First, start up the design and development with a Java interface file or WSDL XML file to generate Web services skeleton and stub, without the existing Web services implementation. Next, develop implementation of Web services on server site and deploy the Web services on Web services server. Finally, develop Web services client on the basis of the Web services stub generated in the first step to access the Web services deployed. If we start off a Web service design with a new WSDL, which is called “green field” design, a Java interface will be generated from the WSDL and rest of the development procedure will be the same.
- *Bottom-Up Design* First, start the design and development of the Web services with an existing Web service such as a Java source file. Next, generate and deploy Web services with different deployment options. Finally, develop the Web services client to get service from the deployed Web services.

Figure 8.3 shows a diagram of AXIS Web services development.

8.2.2 Web Services Development Based on an Existing Web Service Source Code

We will use the same simple temperature conversion example used before to demonstrate how to develop a Web service in Bottom-Up mode. The drop-in simple detail deployment options are discussed here and custom deployment will be discussed in the Web services deployment section.

The Convert Web services has a single method called `toFahrenheit`, taking a Celsius temperature degree as an argument and converting it to a Fahrenheit degree. This

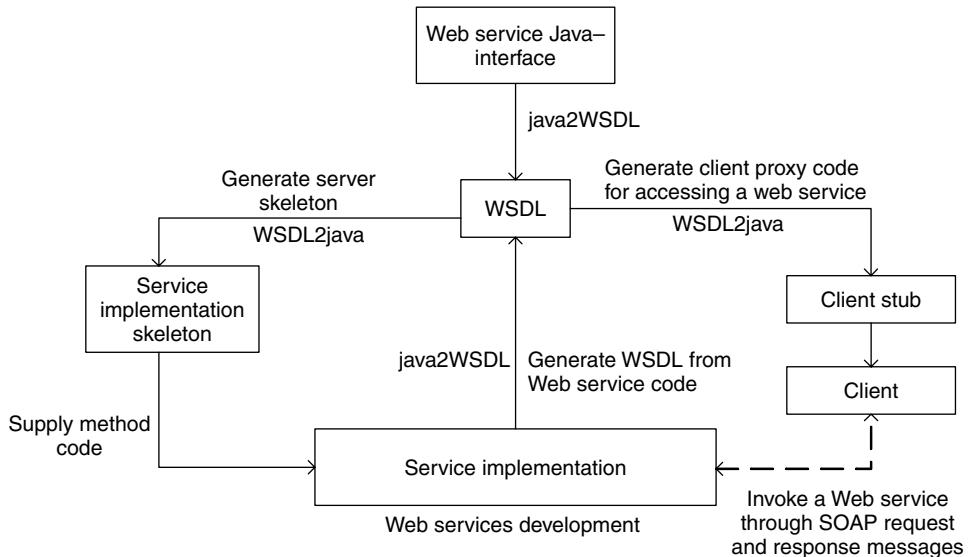


FIGURE 8.3. AXIS Web services development diagram.

example demonstrates how easy it is to create a Web service, deploy a Web service that will programmatically be ready, and develop a Web services client to access this Web service. The detail installation and configuration of AXIS Web services runtime will be discussed in the practice lab section.

Step 1: Get a Web services implementation Java source file.

```
//Convert.java

public class Convert {
    public Convert() { }
    public String toFahrenheit(String pCentigrade) {
        double pCen = Double.parseDouble(pCentigrade);
        double fah = 32 + pCen*9/5;
        return "" + fah;
    }
}
```

Step 2: Deploy the Web services by an easy drop-in deployment.

To deploy this Web service, just copy and paste the `Convert.java` file into the `webapps` directory under `TOMCAT_HOME` directory and change its extension from .Java to .jws. That is all. The Web service is ready to use. The Web services AXIS runtime engine will compile it automatically for the client when the client invokes this Web service. But this type of deployment only works with an available Java source file and not for the bytecode Java class file.

This is the time for us to check the WSDL definition-generated AXIS Web services engine by typing `http://localhost:8080/axis/Convert.jws?wsdl` in the address block of the Internet Explorer. The WSDL definition is shown as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:myDirectory"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
  ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:impl="urn:myDirectory-impl"
  xmlns:intf="urn:myDirectory"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:message name="toFahrenheitResponse">
    <wsdl:part name="return" type="SOAP-ENC:string"/>
  </wsdl:message>

  <wsdl:message name="toFahrenheitRequest">
    <wsdl:part name="in0" type="SOAP-ENC:string"/>
  </wsdl:message>

  <wsdl:portType name="Convert">

    <wsdl:operation name="toFahrenheit" parameterOrder="in0">
      <wsdl:input message="intf:toFahrenheitRequest"/>
      <wsdl:output message="intf:toFahrenheitResponse"/>
    </wsdl:operation>

  </wsdl:portType>

  <wsdl:binding name="ConvertSoapBinding" type="intf:Convert">

    <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="toFahrenheit">

      <wsdlsoap:operation soapAction="" />

      <wsdl:input>

        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:myDirectory" use="encoded"/>

      </wsdl:input>

      <wsdl:output>

        <wsdlsoap:body
          encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:myDirectory" use="encoded"/>

      </wsdl:output>
    
```

```

    </wsdl:operation>

    </wsdl:binding>

    <wsdl:service name="ConvertService">

        <wsdl:port binding="intf:ConvertSoapBinding"
            name="Convert">

            <wsdlsoap:address
                location=
                    "http://localhost:8080/axis/services/Convert"/>

        </wsdl:port>

    </wsdl:service>

</wsdl:definitions>

```

Step 3: Develop a Java client for this Web services component.

ConvertClient.java

```

package myDirectory;

import java.net.URL;
import org.apache.axis.client.Service;
import org.apache.axis.client.Call;
import org.apache.axis.encoding.XMLType;
import javax.xml.rpc.ParameterMode;

public class ConvertClient {

    public ConvertClient() {}

    public static void main (String args[]) {
        try {
            // EndPoint URL for the convert Web service
            String endpointURL =
                "http://localhost:8080/axis/Convert.jws";
            String methodName = "toFahrenheit";
            Service service = new Service();
            // Create Web service object of Call available in client
            // package of API
            Call call = (Call) service.createCall();
            //Set the endPoint URL
            call.setTargetEndpointAddress(new
                java.net.URL(endpointURL));
            //Set the methodname to invoke --- toFahrenheit
            call.setOperationName(methodName);
            //Set three parameters: a user-defined name of the
            //parameter, the parameter data type, and kind of

```

```
//parameter - input only, output only, input/output
call.addParameter("celsius", XMLType.XSD_STRING,
                  ParameterMode.PARAM_MODE_IN);
//set return type of the method
call.setReturnType(XMLType.XSD_STRING);

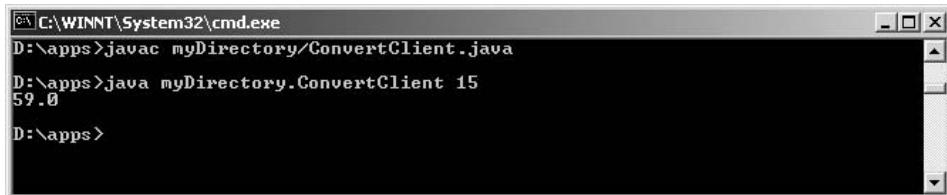
//Setup the command line argument to be passed as input
//parameter to the convert Web service

String result = (String) call.invoke( new Object[] {
    args[0] } );
//Print out the result
System.out.println(result);
}
catch (Exception e) {
    System.err.println(e.toString());
}
}
```

Step 4: Test the Web services.

Compile the `ConvertClient.java` file into a class file and run the client class file with a command line argument of Celsius degree 15 and expected converted Fahrenheit degree is 59, as shown below:

D:\apps>java myDirectory.ConvertClient 15



This is a Java Web services implementation and its client is also programmed in Java. We can also have a Web service implemented in Java and client in .NET or Web services in .NET and client in Java. The interoperability examples will be shown in the section of practice labs.

8.2.3 Web Services Development without Prior Application

Let us start with a Web services definition in Java, which just specifies a package and an interface with its method declaration without any detailed implementation.

Step 1: Provide a Java interface in `Convert.java`.

```

package convert;

public interface Convert
{
    public String toFahrenheit(String pCentigrade);
}

```

Step 2: Create a WSDL interface from this Java interface by Java2WSDL.

```

D:\apps\myDirectory>java org.apache.axis.wsdl.Java2WSDL -o
convert.wsdl -l
“http://localhost : 8080/axis/services/Convert” -n “urn:myDirectory”
-p“convert” “urn:myDirectory” convert.Convert

```

Here -o indicates the output for the WSDL filename, -l specifies Web service location URL, -n specifies the target namespace, and -p provides a mapping from Java package to namespace to ensure that all classes in the specified package will be mapped to the namespace. The convert.wsdl is generated now, which is shown in the previous section.

Step 3: Create Bindings using WSDL2Java.

```

D:\apps\myDirectory>java org.apache.axis.wsdl.WSDL2Java -o. -d session
-s -Nurn:myDirectory convert convert.wsdl

```

WSDL2Java takes a number of options. -o specifies the output directory such as current working directory “.” in this example. -d specifies the deployment scope such as request, session, and application. -N specifies the mapping from namespace to package. -s option generates all server-side skeleton and binding, and a WSDL XML file deploys. wsdd and undeploy.wsdd. WSDL2Java also generates client-side stub proxy.

We will discuss all the files generated by WSDL2Java.

- WSDL2Java generates <typename>.java for each <type> definition in WSDL XML document.
- WSDL2Java generates an interface Java file <portTypename>.java for each <portType> in WSDL. The file Convert.java is generated corresponding to the portType “Convert.” This file plays a role of an interface in Java.

Here is the list of this file.

```

/**
 * Convert.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package convert;

public interface Convert extends java.rmi.Remote {

```

```

        public java.lang.String toFahrenheit(java.lang.String
            in0) throws
java.rmi.RemoteException;
}

```

- WSDL2Java also generates a <portName>SOAPBindingStub.java for each <binding> in WSDL specification. This file implements SDI and is used at client side as a stub. The following is the list of this stub template. ConvertSoapbinding is the name of binding in WSDL.

```

/*
 * ConvertSoapBindingStub.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package convert;

public class ConvertSoapBindingStub extends
    org.apache.axis.client.Stub
implements convert.Convert {
    private java.util.Vector cachedSerClasses = new
        java.util.Vector();
    private java.util.Vector cachedSerQNames = new
        java.util.Vector();
    private java.util.Vector cachedSerFactories = new
        java.util.Vector();
    private java.util.Vector cachedDeserFactories = new
        java.util.Vector();

    public ConvertSoapBindingStub() throws
        org.apache.axis.AxisFault {
        this(null);
    }

    public ConvertSoapBindingStub(java.net.URL endpointURL,
        javax.xml.rpc.Service service) throws
        org.apache.axis.AxisFault {
        this(service);
        super.cachedEndpoint = endpointURL;
    }

    public ConvertSoapBindingStub(javax.xml.rpc.Service service)
        throws
        org.apache.axis.AxisFault {
        try {
            if (service == null) {
                super.service = new
                    org.apache.axis.client.Service();
            } else {

```

```

        super.service = service;
    }
}
catch(Exception t) {
    throw org.apache.axis.AxisFault.makeFault(t);
}
}

private org.apache.axis.client.Call getCall() throws
java.rmi.RemoteException {
try {
    org.apache.axis.client.Call call =
        (org.apache.axis.client.Call)
        super.service.createCall();
    if (super.maintainSessionSet) {
        call.setMaintainSession(super.maintainSession);
    }
    if (super.cachedUsername != null) {
        call.setUsername(super.cachedUsername);
    }
    if (super.cachedPassword != null) {
        call.setPassword(super.cachedPassword);
    }
    if (super.cachedEndpoint != null) {
        call.setTargetEndpointAddress
            (super.cachedEndpoint);
    }
    if (super.cachedTimeout != null) {
        call.setTimeout(super.cachedTimeout);
    }
    java.util.Enumeration keys =
        super.cachedProperties.keys();
    while (keys.hasMoreElements()) {
        String key = (String) keys.nextElement();
        call.setProperty(key,
                         super.cachedProperties.get(key));
    }
    // All the type mapping information is registered
    // when the first call is made.
    // The type mapping information is actually
    // registered in
    // the TypeMappingRegistry of the service, which is
    // the reason why registration is only needed for the
    // first call.
    if (firstCall()) {
call.setEncodingStyle(org.apache.Constants.
URI_SOAP_ENC);
        for (int i = 0; i < cachedSerFactories.size();
            ++i) {
            Class cls = (Class) cachedSerClasses.get(i);
            javax.xml.rpc.namespace.QName qName =
                (javax.xml.rpc.namespace.QName)
            cachedSerQNames.get(i);
        }
    }
}
}

```

```

        Class sf = (Class)
                  cachedSerFactories.get(i);
        Class df = (Class)
                  cachedDeserFactories.get(i);
        call.registerTypeMapping(cls, qName, sf, df,
                                false);
    }
}
return call;
}
catch (Throwable t) {
    throw new org.apache.axis.AxisFault("Failure trying
        to get the Call object", t);
}
}

public java.lang.String toFahrenheit(java.lang.String in0)
throws java.rmi.RemoteException{
if (super.cachedEndpoint == null) {
    throw new org.apache.axis.NoEndPointException();
}
org.apache.axis.client.Call call = getCall();
javax.xml.rpc.namespace.QName pQName = new
javax.xml.rpc.namespace.QName("", "in0");
call.addParameter(pQName, new
javax.xml.rpc.namespace.QName("http://schemas.xmlsoap.org/
    soap/encoding
/", "string"), javax.xml.rpc.ParameterMode.PARAM_MODE_IN);
call.setReturnType(new
javax.xml.rpc.namespace.QName("http://schemas.xmlsoap.org/
    soap/encoding/", "string"));
call.setUseSOAPAction(true);
call.setSOAPActionURI("");
call.setOperationStyle("rpc");
call.setOperationName(new
javax.xml.rpc.namespace.QName("urn:myDirectory",
    "toFahrenheit"));

Object resp = call.invoke(new Object[] {in0});

if (resp instanceof java.rmi.RemoteException) {
    throw (java.rmi.RemoteException)resp;
}
else {
    return (java.lang.String) resp;
}
}
}

```

- WSDL2Java also generates a server-side implementation skeleton Java file corresponding to this `ConvertSoapBinding` binding specification in the WSDL. For each `<service>` specification, WSDL2Java also generates a `<service>Service`.

java and <service>ServiceLocator.java files. In our example, a *ConvertService.java* is generated corresponding to the Convert name in port element of service element. It is a Java interface declaring get method for each port defined in the service.

```
/**
 * ConvertService.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package convert;

public interface ConvertService extends javax.xml.rpc.Service {
    public String getConvertAddress();

    public convert.Convert getConvert() throws
        javax.xml.rpc.ServiceException;

    public convert.Convert getConvert(java.net.URL portAddress)
        throws javax.xml.rpc.ServiceException;
}
```

A ConvertServiceLocator.java file is generated, which implements the above interface. It is used to locate an implementation of the service and creates and returns an instance of the stub.

```
/**
 * ConvertServiceLocator.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package convert;

public class ConvertServiceLocator extends
    org.apache.axis.client.Service
implements convert.ConvertService {

    // Use to get a proxy class for Convert
    private final java.lang.String Convert_address =
        "http://localhost:8080/axis/services/Convert";

    public String getConvertAddress() {
        return Convert_address;
    }

    public convert.Convert getConvert() throws
        javax.xml.rpc.ServiceException {
        java.net.URL endpoint;
```

```

        try {
            endpoint = new java.net.URL(Convert_address);
        }
        catch (java.net.MalformedURLException e) {
            return null;
            // unlikely as URL was validated in WSDL2Java
        }
        return getConvert(endpoint);
    }

    public convert.Convert getConvert(java.net.URL portAddress)
        throws javax.xml.rpc.ServiceException {
        try {
            return new
                convert.ConvertSoapBindingStub(portAddress, this);
        }
        catch (org.apache.axis.AxisFault e) {
            return null;
        }
    }
}

```

Step 4: Customize the skeleton template ConvertSoapBindingImpl.java generated in Step 3 to fill in the implementation code.

```

/**
 * ConvertSoapBindingImpl.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package convert;

public class ConvertSoapBindingImpl implements convert.Convert
{
    public java.lang.String toFahrenheit(java.lang.String in0)
        throws java.rmi.RemoteException {
        double pCen = Double.parseDouble(in0);
        double fah = 32 + pCen*9/5;
        return "" + fah;
    }
}

```

Step 5: Compile the Java files generated.

D:\apps\myDirectory>javac convert/*.java

Step 6: Make a jar file to be deployed.

D:\apps\myDirectory>jar cvf conv.jar convert/*.class

Step 7: Copy the jar file to a proper directory; set CLASSPATH to include that directory.

```
D:\apps\myDirectory>copy conv.jar
D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\
```

Step 8: Deploy Web services with WSDD. The detail contents of WSDD and AdminClient utility are discussed in Web services deployment section.

```
D:\apps\myDirectory\convert>java org.apache.axis.client.AdminClient
deploy.wsdd
```

Step 9: Develop a Web services client testing program ConvertTest.java, and compile and run it for testing.

```
// ConvertTest.java

package convert;

public class ConvertTest

{
    public static void main(String args[]) throws Exception

    {
        convert.ConvertService service =
            new convert.ConvertServiceLocator();

        convert.Convert conve = service.getConvert();
        System.out.println("The result is " +
                           conve.toFahrenheit(args[0]));

    }
}
```

```
D:\apps\myDirectory>javac convert/ConvertTest.java
D:\apps\myDirectory>java convert.ConvertTest 17
```

In this section, we discussed the concept of Java Web services implemented in AXIS. Other implementations such as .NET Web service and Web service interoperability will be covered in the practice lab section.

8.3 CONNECTION MODEL OF WEB SERVICES

8.3.1 Interactions Between Web Services

8.3.1.1 Synchronous Interaction Versus Asynchronous Interactions Web services invocation is either synchronous or asynchronous. In synchronous interaction, a client sends a Web services request and halts its operation while waiting for a response. If the

service takes a tremendous computation time to complete, an asynchronous interaction must be used instead. The synchronous interaction is the default interaction in Web services operations. Another reason is that in some cases there is no backward channel available for the response to come back synchronously such as e-mail response to a HTTP request.

There are four `portType` of operations in WSDL interface definitions. They can be divided into two groups: asynchronous message-based operations and synchronous RPC-based operations.

Asynchronous message-based operation is a single-message passing operation, which can be one-way incoming notification and one-way outgoing notification. One-way message passing operation never expects any response right away. The message may be a signal, a notification, or data to be processed by the target of the message. The two-way request/response operation is an RPC-based operation with a combination of incoming and outgoing message. The source of the message sends out an outgoing SOAP message and expects to get an incoming response SOAP message right away. In the out/in mode, the target of message gets incoming message and responds to an outgoing message in the out/in mode. Obviously, they are synchronous operations.

The basic Web services connection is established between two endpoints of Web services. The source `webservice1` has an outgoing message port, which connects a same type incoming message port of target `webservice2`. In Figure 8.4, there are two separate unrelated one-time channels (not permanent) between `webservice1` and `webservice2`. The `webservice2` provides an outgoing message port, which connects to a same type incoming message port of `webservice1`.

The `webservice2` may process data according to the message received from `webservice1` or process the message itself and then forward it to third Web services.

This message-based operation is a typical asynchronous communication style that a client of such Web services does not need to wait for any reply back from the Web services it requests and just simply continue its own control flow.

RPC-based request/response message exchange takes place between two endpoints of Web services where `webservice1` has an out/in port and `webservice2` has an in/out port so that `webservice1` can send a SOAP request to the in port of `webservice2` and `webservice2` responds a SOAP message via its out port to the in port of `webservice1`. The sequence of incoming message before outgoing messages in the definition of `portType` in WSDL specifies an in/out `portType` operation, which is a service provider operation. The sequence of outgoing message before incoming messages in the definition of `portType` in WSDL specifies an out/in `portType` operation, which is a service request operation. RPC-based style service is a typical synchronous communication style service that a client must wait till the result of the invoked method of Web services is received.

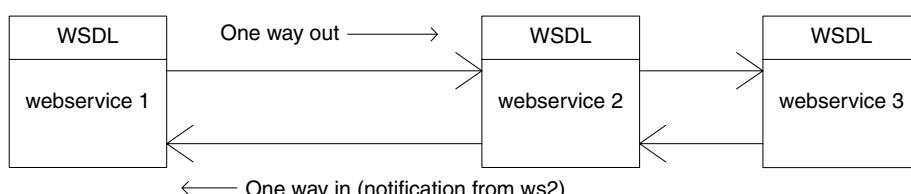


FIGURE 8.4. Asynchronous message-based connection between Web services.

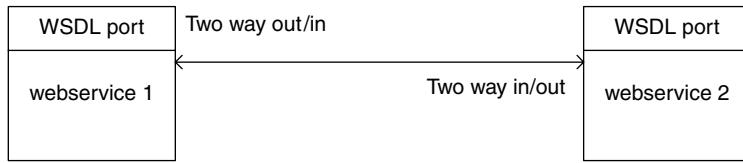


FIGURE 8.5. Synchronous RPC-based connections between Web services.

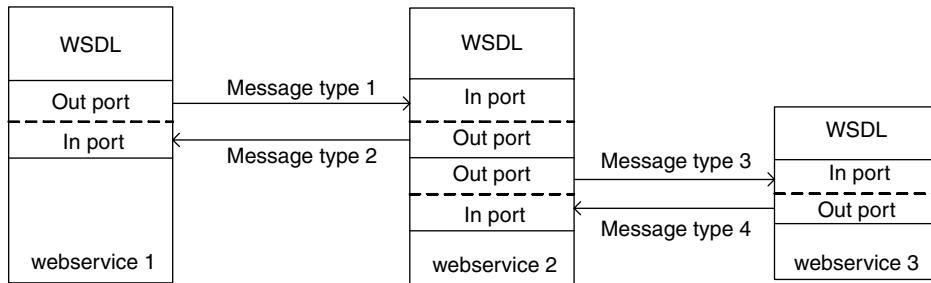


FIGURE 8.6. Synchronous RPC-based connection between ports of Web services.

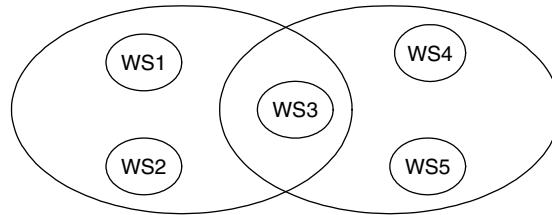
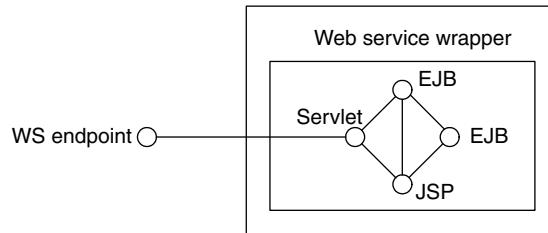
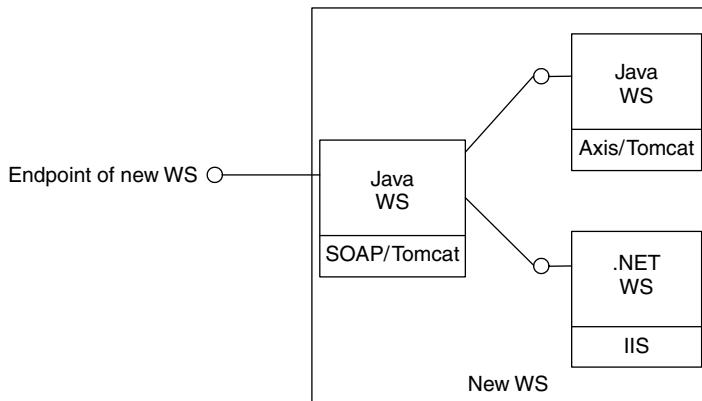
Figures 8.5 and Figure 8.6 depict the synchronous connections between two Web services.

8.3.1.2 Static Invocation Versus Dynamic Invocation There are two different ways for a Web services client to invoke a method provided by a Web service, static invocation and dynamic invocation. In static invocation mode, the client generates a proxy stub during the development time or compilation time. The WSDL of the Web services must be available at that time since this WSDL describes the signatures and all necessary information for the method invocation. The client will invoke the remote method provided by the Web services via invocation of the method in proxy. The client does not need to parse the WSDL when the method is invoked. So the method invocation is very efficient. The downside of it is that the WSDL must be unchanged; otherwise, the proxy must be generated again, which is different from dynamic invocation.

In dynamic invocation mode, the client does not need any static proxy stub. The proxy stub is dynamically generated at runtime using dynamic invocation APIs instead of development time. The advantage is that any WSDL changes after the client development will not affect the Web services invocation. The client will retrieve and interpret the WSDL at runtime and dynamically construct a call. Although it is not as efficient as static invocation in terms of method invocation, it is much more flexible.

8.3.2 Web Services Composition Versus Web Service Conversation

So far, we have discussed the styles of Web services conversations among Web services and from client to Web services. We can also compose a new Web service from many existing Web services. In order to make Web service truly useful in A2A, B2B, there must be a way to combine and coordinate collections of Web services, to glue them together, so that they can be integrated to support a large-scale real-world

**FIGURE 8.7.** Web services composition.**FIGURE 8.8.** Web services wrappers for a J2EE enterprise application.**FIGURE 8.9.** Web services wrappers (containment) of Web services composition.

business enterprise applications. A new Web service has its own identification, which is different from Web services interaction or conversation. One Web service can be composed in many other Web services and one Web service may consist of many Web services. There is a many-to-many relationship between a Web service and Web services composition. There are two composed Web services in Figure 8.7, where both of them use ws3 as a part of their Web services. We can see that a Web services composition is a Web services authoring to combine or glue multiple Web services into a new large Web service. Web services Flow Language (WSFL) may be used to determine the sequence of subprocess of Web services that form a business process and flow of information. Some people call Web services composition as Web services orchestration.

A Web services composition can be implemented by a containment construction, in which the outer Web service holds a reference to one or many inner Web services and outer Web service forwards a Web service request to an inner Web service. The `endPoint` of inner Web services is behind the scene. An example of containment construction is the Web service `endPoint` for stateless session EJB shown in Figure 8.8.

A Web services composition can also be implemented by an aggregation construction, in which an inner Web services `endPoint` reference is handed out directly to the outer Web services client. Figure 8.9 shows both of these compositions, which are hardwired at design time.

8.4 WEB SERVICES COMPONENT DEPLOYMENT

Web services components need to be deployed just like any other type components. In Section 8.2, we have discussed a simple drop-in deployment, but it has many restrictions such as the availability of the source code of Web services. In this section, we discuss the Web services custom deployment, which is much more flexible than a drop-in deployment. We demonstrate this deployment using the same example of temperature conversion that we used before.

This type deployment requires a Web services Deployment Descriptor (WSDD) XML document to deploy a Web service on-line.

Assume we only have Web services class files on hand. There are two ways to generate WSDD XML files. One way is to create (write) WSDD from scratch and the other way is to generate a WSDD by AXIS tools, which generate WSDL from class files and then generate WSDD from WSDL.

First let us take a look at a WSDD XML document `deploy.wsdd` for our temperature conversion Web services, which is generated by the tools.

```
<!-- Use this file to deploy some handlers/chains and services -->
<!-- Two ways to do this:                                     -->
<!--   java org.apache.axis.utils.Admin deploy.wsdd          -->
<!--       from the same directory that the Axis engine runs -->
<!-- or                                                 -->
<!--   java org.apache.axis.client.AdminClient deploy.wsdd    -->
<!--       after the axis server is running                  -->

<deployment
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <!-- Services from ConvertService WSDL service -->

  <service name="Convert" provider="java:RPC">
    <parameter name="className"
              value="convert.ConvertSoapBindingSkeleton"/>
    <parameter name="allowedMethods" value="toFahrenheit"/>

  </service>
</deployment>
```

The root element in this WSDD is a deployment element, which has a subelement *service*. The root element defines a default namespace of WSDD, a namespace for Java provider, and an XML Schema instance namespace. The *service* element defines the name of the Web services used to provide a unique name for the Web services and type of provider. The default types of provider may be RPC-based type or document-based types. RPC type works for request/response synchronous communication and document type works for message passing asynchronous communication. AXIS also supports EJB-based type provider and COM provider, depending on the kind the Web services. The child elements, which are both *parameters*, describe the class available to the service and the methods to be accessed. The name/value pairs tell the names of the classes and the names of methods for this deployed Web services.

The deployment can be implemented by command line utility or by Apache- AXIS Admin utility at <http://<URL>:<port>/axis/index.html>, for example, <http://localhost:8080/axis/index.html> to deploy or check the deployed Web service. We focus on the command line Web service deployment in this section.

The deployment command line is shown in the following:

```
D:apps\myDirectory\convert>java org.apache.axis.client.AdminClient
deploy.wsdd
```

Before we run this command, we need to make sure the Tomcat is up and running and *deploy.wsdd* is available in this directory. *AdminClient* takes a WSDD XML file as its arguments to deploy the Web service or takes an argument of *list -l AdminServiceURL* to list all deployed Web services in AXIS. The *AdminServiceURL* is the URL of *AdminService*, which is typically accessible at <http://localhost:8080/axis/services/AdminService>.

In order to remove or undeploy a Web service from the AXIS server, an *undeploy.wsdd* is needed. The *undeploy.wsdd* can be created manually or generated by AXIS utility *WSDL2Java*, which generates both of *deploy.wsdd* and *undeploy.wsdd* at the same time.

The following is the *undeploy.wsdd* file for *Convert* Web services.

```
<!-- Use this file to undeploy some handlers/chains and services
-->
<!-- Two ways to do this:
<!--   java org.apache.axis.utils.Admin undeploy.wsdd
-->
<!--       from the same directory that the Axis engine runs
-->
<!-- or
<!--   java org.apache.axis.client.AdminClient undeploy.wsdd
-->
<!--       after the axis server is running
-->

<undeployment
  xmlns="http://xml.apache.org/axis/wsdd/">

  <!-- Services from ConvertService WSDL service -->

  <service name="Convert" />
</undeployment>
```

We can remove the Web services we deployed by the following command:

```
D:\apps\myDirectory\convert>java org.apache.axis.client.AdminClient
deploy.wsdd
```

8.5 EXAMPLES AND LAB PRACTICE

In this section, we will demonstrate Web services design, development, deployment, and Web services client application. The interoperability of Web services will also be explored in the practice labs.

8.5.1 Lab 1: Apache AXIS Web Service Engine Installation

In order to practice all labs in this chapter, we need to have the following:

1. Tomcat 4.0

Follow the link to download `jakarta-tomcat-4.0.6.zip` or newest version <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/> and

Extract the package to the directory you prefer, for example, `D:\apps\jakarta-tomcat-4.0.6`.

2. Apache AXIS

Go to <http://ws.apache.org/axis/dist/beta1/> and download `xml-axis-beta1.zip` and extract the package to a directory you prefer such as `D:\apps\xml-axis-beta1`.

3. Java Development Kit version 1.4.0

Follow the links to download JDK from <http://java.sun.com/j2se/> and install it in `<root-directory>\j2sdk1.4.0` such as `D:\j2sdk1.4.0` or newest version.

4. Microsoft Visual Studio .NET for .NET Web services

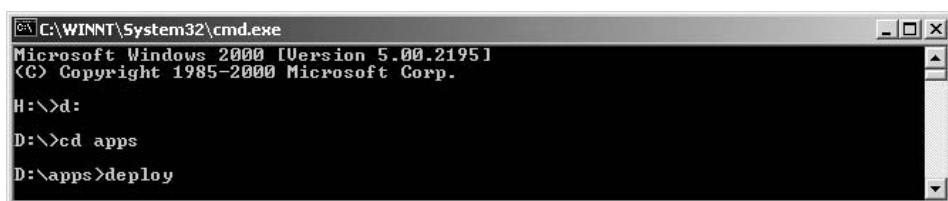
After downloading all the above files, we can start installations.

Step 1: Copy the `webapps/axis` directory from the `xml-axis` distribution into `Jakarta-tomcat-4.0.6\webapps` directory. For example, `D:\apps\jakarta-tomcat-4.0.6\webapps\axis`

Step 2: Copy the `xerces.jar` file from `D:\apps\jakarta-tomcat-4.0.6\common\lib` to `D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib`

Step 3: Configure the environment and deploy the AXIS by the command:

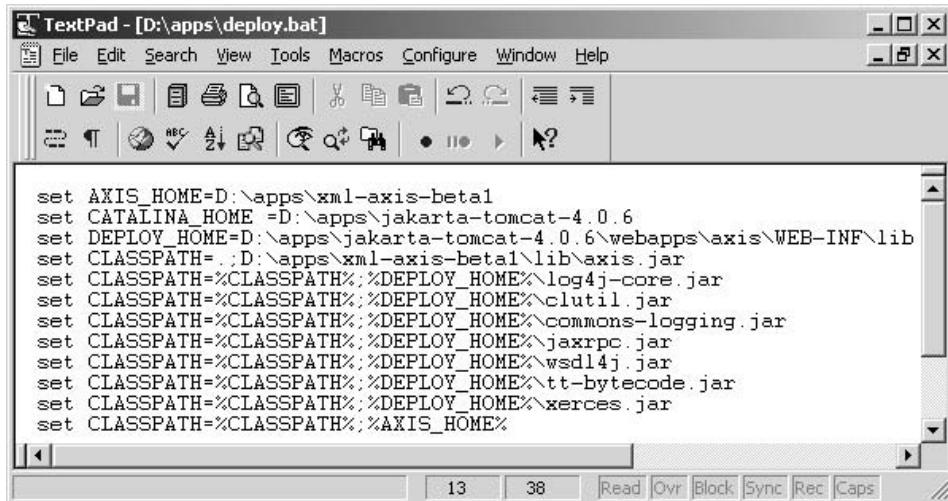
```
D:\apps>deploy
```



```

set AXIS_HOME=D:\apps\xml-axis-beta1
set CATALINA_HOME =D:\apps\jakarta-tomcat-4.0.6
set DEPLOY_HOME=D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-
INF\lib
set CLASSPATH=.;D:\apps\xml-axis-beta1\lib\axis.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\log4j-core.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\clutil.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\commons-logging.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\jaxrpc.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\wsdl4j.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\tt-bytecode.jar
set CLASSPATH=%CLASSPATH%;%DEPLOY_HOME%\xerces.jar
set CLASSPATH=%CLASSPATH%;%AXIS_HOME%

```



8.5.2 Lab 2: Java Web Service/Java Client with Drop-in Deployment

The Web services practiced in this lab is the temperature conversion Web services that we discussed before. In this lab, we will go through a Java Web services development with an existing Web services Java source code and a Java client. The Web service is deployed by drop-in deployment. This Web services design and development is in a bottom-up model.

Step 1: Copy the Convert.java file to D: \apps\jakarta-tomcat-4.0.6\ webapps\axis and rename it as Convert.jws.

```

// Convert.java
public class Convert {

    public Convert() {
    }
    public String toFahrenheit(String pCentigrade) {
        double pCen = Double.parseDouble(pCentigrade);

```

```

        double fah = 32 + pCen*9/5;
        return "" + fah;
    }
}

```

Step 2: To test our Web services, we develop a `ConvertClient.java` file, which is a client of our Web services. First, make sure that Tomcat is up and running on your machine on port 8080.

```

//ConvertClient.java
package myDirectory;

import java.net.URL;
import org.apache.axis.client.Service;
import org.apache.axis.client.Call;
import org.apache.axis.encoding.XMLType;
import javax.xml.rpc.ParameterMode;

public class ConvertClient {

    public ConvertClient() {}

    public static void main (String args[]) {
        try {
            // EndPoint URL for the Web Service
            String endpointURL =
                "http://localhost:8080/axis/Convert.jws";
            String methodName = "toFahrenheit";
            Service service = new Service();
            Call call = (Call) service.createCall();
            //Set the endPoint URL
            call.setTargetEndpointAddress(new
                java.net.URL(endpointURL));
            //Set the methodname to invoke - toFahrenheit
            call.setOperationName(methodName);
            call.addParameter("Celsius", XMLType.XSD_STRING,
                ParameterMode.PARAM_MODE_IN);
            call.setReturnType(XMLType.XSD_STRING);

            /*Setup the Parameter of temperature in Celsius to be
             passed as input parameter to the Convert Web Service*/

            String result = (String) call.invoke( new Object[] {
                args[0] } );
            //Print out the result
            System.out.println(result);
        }
        catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}

```

Run the Web services client program:

```
D:\apps>java myDirectory.ConvertClient 15
```

The converted temperature 59 in Fahrenheit is shown on the screen.

```
C:\WINNT\System32\cmd.exe
D:\apps>javac myDirectory(ConvertClient.java
D:\apps>java myDirectory.ConvertClient 15
59.0
D:\apps>
```

8.5.3 Lab 3: Demonstration on Web Services SOAP Request and Web Services SOAP Response Using TCP Monitor

This lab is a continuation of Lab 1 to demonstrate the SOAP request and response in Web services. AXIS provides a tool called `tcpmon`, which can intercept the SOAP messages during the Web services processing to show the XML SOAP request and response.

Step 1: Let `tcpmon` listen on the port 9000 to check the activities on the HTTP port 8080. It will start up a `TcpMonitor` window.

```
D:\apps>java org.apache.axis.utils.tcpmon 9000 localhost 8080
```

```
Select C:\WINNT\System32\cmd.exe - Java org.apache.axis.utils.tcpmon 9000 localhost 8080
INF\lib\commons-logging.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\jakarta-jaxrpc.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\wsdl4j.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\tt-bytocode.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\xerces.jar;D:\apps\xml-axis-beta1\lib\javac myDirectory.ConvertClient.java
D:\apps>java myDirectory.ConvertClient 15
59.0
D:\apps>Java org.apache.axis.utils.tcpmon 9000 localhost 8080
```

Step 2: Change one line in `ConvertClient.java`. Replace the line

```
String endpointURL = "http://localhost:8080/axis/Convert.jws";
```

with the following line.

```
String endpointURL = "http://localhost:9000/axis/Convert.jws";
```

Step 3: Open another command line prompt window.

Set CLASSPATH again, then recompile and run `ConvertClient`.

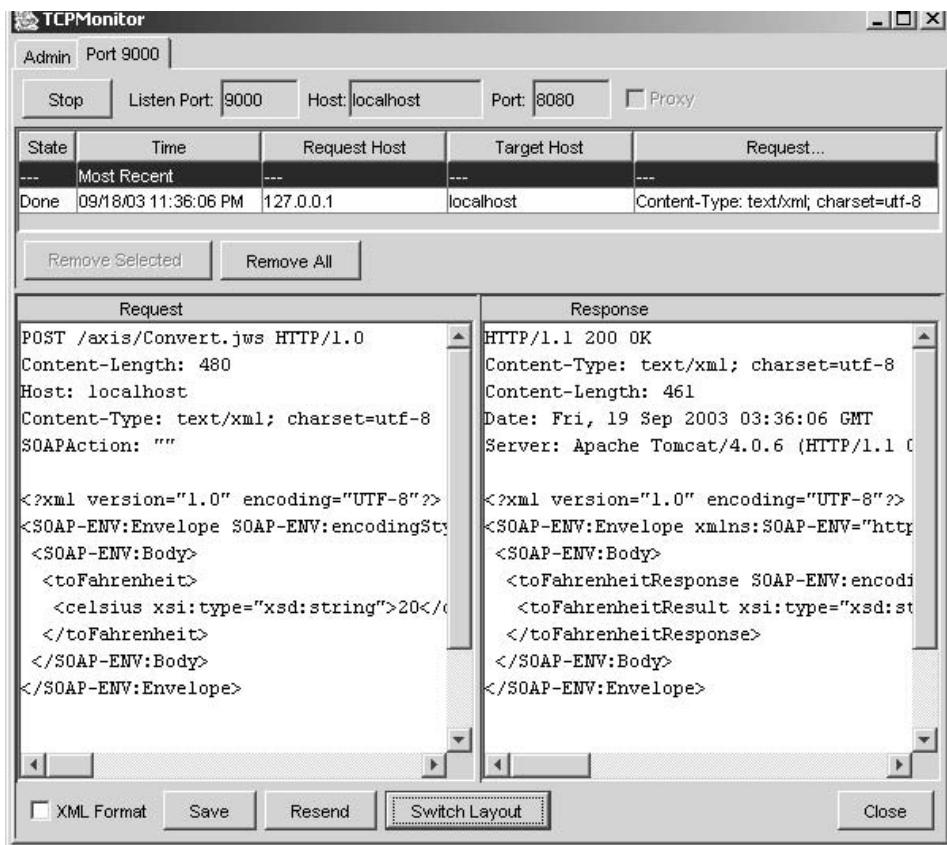
```
D:\apps>javac myDirectory(ConvertClient.java
D:\apps>java myDirectory.ConvertClient 20
```

We will see the result of this Web services request in the following screen snapshot:



```
C:\WINNT\System32\cmd.exe
b\jaxrpc.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\wsdl4j.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\tt-bytecode.jar;D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\xerces.jar;D:\apps\xml-axis-beta1
D:\apps>javac myDirectory.ConvertClient.java
D:\apps>java myDirectory.ConvertClient 20
68.0
D:\apps>
```

The SOAP Request and Response are shown in TCPMonitor window.



The complete SOAP request and SOAP response messages for this Web services are shown in the following:

SOAP Request:

```
POST /axis/Convert.jws HTTP/1.0
Content-Length: 480
Host: localhost
Content-Type: text/xml; charset=utf-8
```

```
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <toFahrenheit>
      <celsius xsi:type="xsd:string">20</celsius>
    </toFahrenheit>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 461
Date: Fri, 19 Sep 2003 03:36:06 GMT
Server: Apache Tomcat/4.0.6 (HTTP/1.1 Connector)
```

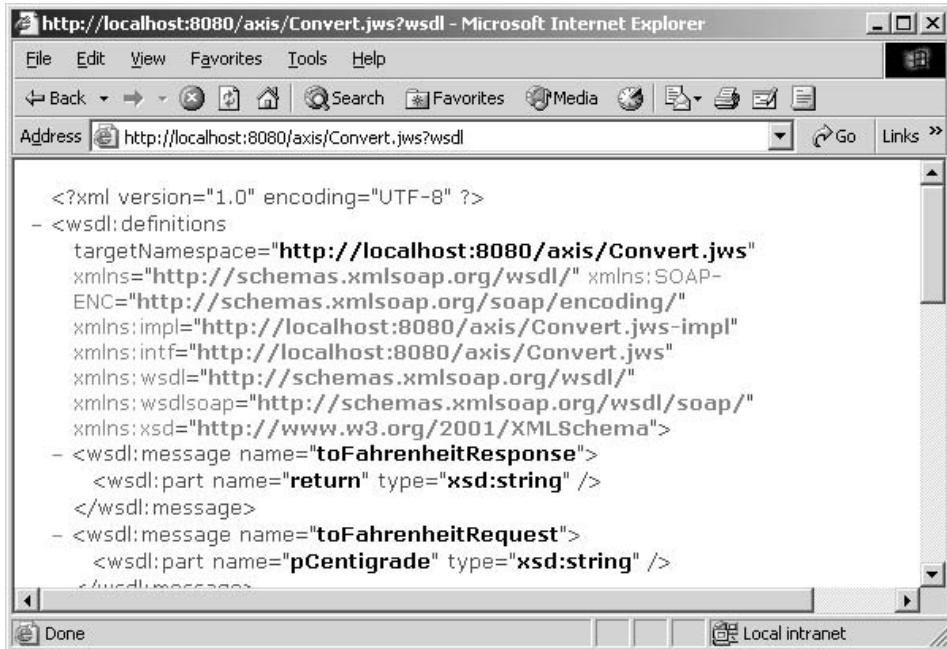
```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <toFahrenheitResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <toFahrenheitResult
                xsi:type="xsd:string">68.0</toFahrenheitResult>
        </toFahrenheitResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

8.5.4 Lab 4: Web Services Interoperability with Java Web Services and .NET Client

So far, we have seen the Web services implementation in Java and service request coming from a Java client. In this section, we will practice an example of Java Web services with a .NET client. We use the same Java Web services as before except that the client is .NET client. This Web services design and development is done in top-down model.

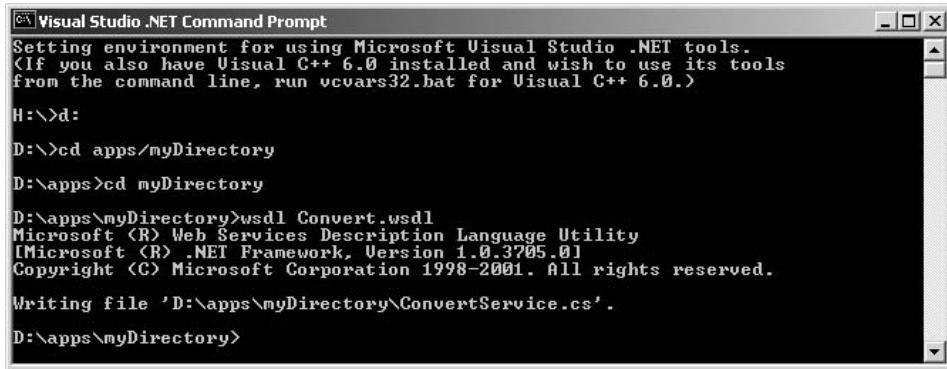
Step 1: Get WSDL specification of Convert Web services.

Type `http://localhost:8080/axis/Convert.jws?wsdl` in the address block of the Microsoft Internet Explorer. A WSDL document is displayed.



Save convert.wsdl to D:\apps\myDirectory directory by clicking on “Save as” from “File” menu bar. Next, open a Visual Studio .NET Command Prompt and use the saved Convert.wsdl to generate a template C# Web service client ConvertService.cs.

D:\apps\myDirectory>wsdl Convert.wsdl



Step 2: Modify the client template ConvertService.cs and add a main method

```
public static void Main( string [] args ) {
    ConvertService service = new ConvertService();
```

```

        string response = service.toFahrenheit(args[0]);
        Console.WriteLine("The result is " + response);
    }
}

```

Step 3: Compile and run the client ConvertService in .NET framework.

```

D:\apps\myDirectory>csc ConvertService.cs
D:\apps\myDirectory>ConvertService 16

```

```

C:\Visual Studio .NET Command Prompt

D:\apps\myDirectory>csc ConvertService.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

D:\apps\myDirectory>ConvertService 16
The result is 60.8
D:\apps\myDirectory>

```

The converted temperature result is shown above.

8.5.5 Lab 5: Web Service Custom Deployment with WSDD

Step 1: Generate a WSDL from a Java interface or class without Web service source code or create a WSDL specification manually. Here is a Java interface definition.

```

package convert;

public interface Convert
{
    public String toFahrenheit(String pCentigrade);
}

```

Step 2: Create WSDL using Java2WSDL.

```

D:\apps\myDirectory>java org.apache.axis.wsdl.Java2WSDL -o
convert.wsdl -l"http://localhost:8080/axis/services/Convert" -n
"urn:myDirectory" -p"convert" "urn:myDirectory" convert.Convert

```

```

C:\WINNT\System32\cmd.exe

D:\apps\myDirectory>java org.apache.axis.wsdl.Java2WSDL -o convert.wsdl -l"http://localhost:8080/axis/services/Convert" -n "urn:myDirectory" -p"convert" "urn:myDirectory" convert.Convert
D:\apps\myDirectory>

```

Step 3: Generate Web services stub and skeleton, and deployment wsdd using WSDL2Java.

```
D:\apps\myDirectory>java org.apache.axis.wsdl.WSDL2Java -o. -d session
-s -Nurn:myDirectory convert convert.wsdl
```

```
D:\apps\myDirectory>java org.apache.axis.wsdl.WSDL2Java -o. -d session -s -Nurn:myDirectory convert convert.wsdl
Unrecognized scope: session. Ignoring it.
D:\apps\myDirectory>_
```

Step 4: Override ConvertSoapBindingImpl.java with implementation of toFahrenheit method and compile it.

```
package convert;

public class ConvertSoapBindingImpl implements convert.Convert {
    public java.lang.String toFahrenheit(java.lang.String in0) throws
        java.rmi.RemoteException {
        double pCen = Double.parseDouble(in0);
        double fah = 32 + pCen*9/5;
        return "" + fah;
    }
}
```

```
D:\apps\myDirectory>javac convert/*.java
```

```
D:\apps\myDirectory>javac convert/*.java
D:\apps\myDirectory>
```

Step 5: Make a Web services Java archive file with an extension .jar for Web service deployment.

```
D:\apps\myDirectory>jar cvf conv.jar convert/*.class
```

```
C:\> C:\WINNT\System32\cmd.exe
D:\apps\myDirectory>jar cvf conv.jar convert/*.class
added manifest
adding: convert/Convert.class<in = 243> <out= 171><deflated 29%>
adding: convert/ConvertService.class<in = 353> <out= 217><deflated 38%>
adding: convert/ConvertServiceLocator.class<in = 993> <out= 555><deflated 44%>
adding: convert/ConvertSoapBindingImpl.class<in = 731> <out= 426><deflated 41%>
adding: convert/ConvertSoapBindingSkeleton.class<in = 2107> <out= 955><deflated 54%>
adding: convert/ConvertSoapBindingStub.class<in = 3827> <out= 1969><deflated 48%>
D:\apps\myDirectory>
```

Copy the jar file to a proper directory; set CLASSPATH to include the directory where the jar file resides.

```
D:\apps\myDirectory>copy conv.jar
D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\
```

```
C:\> C:\WINNT\System32\cmd.exe
D:\apps\myDirectory>copy conv.jar D:\apps\jakarta-tomcat-4.0.6\webapps\axis\WEB-INF\lib\
1 file(s) copied.
D:\apps\myDirectory>
```

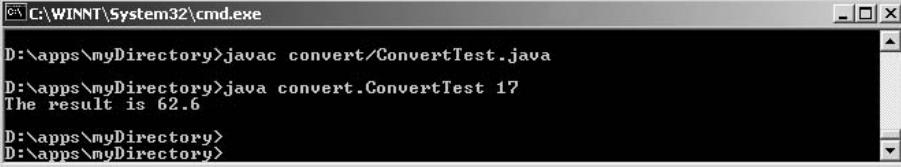
Step 6: Deploy the Web service with WSDD file.

```
D:\apps\myDirectory\convert>java org.apache.axis.client.AdminClient
deploy.wsdd
```

```
C:\> C:\WINNT\System32\cmd.exe
D:\apps\myDirectory\convert>java org.apache.axis.client.AdminClient deploy.wsdd
- Processing file deploy.wsdd
<Admin>Done processing</Admin>
D:\apps\myDirectory\convert>
```

Make a Web services client ConvertTest.java and compile and run that for testing. The source code is listed below.

```
D:\apps\myDirectory>javac convert/ConvertTest.java
D:\apps\myDirectory>java convert.ConvertTest 17
```



```
D:\apps\myDirectory>javac convert/ConvertTest.java
D:\apps\myDirectory>java convert.ConvertTest 17
The result is 62.6
D:\apps\myDirectory>
```

```
package convert;

public class ConvertTest
{
    public static void main(String args[]) throws Exception
    {
        convert.ConvertService service =
            new convert.ConvertServiceLocator();

        convert.Convert conve = service.getConvert();
        System.out.println("The result is " +
            conve.toFahrenheit(args[0]));
    }
}
```

8.5.6 Lab 6: .NET Web Services

In this lab, we will practice a .NET Web services implemented in .NET C#. We also show the result of a Web services request in an XMP format before being further processed.

Step 1: Make Convert.asmx file and save in C:\Inetpub\wwwroot directory. Assume a Microsoft IIS Web server is available now. The wwwroot directory is a root directory for www population.

```
// Convert.asmx
<%@ Web services Language="C#" Class="Convert" %>

using System;
using System.Web.Services;

public class Convert : Web services {
    [WebMethod] public float toFahrenheit(String pCentigrade) {
        float pFen = float.Parse(pCentigrade);
        float fah = 32 + pFen*9/5;
        return fah;
    }
}
```

Step 2: Type `http://localhost/Convert.asmx` in internet explorer address box. A Web services Convert window appears and we can click the link of Web services method `toFahrenheit` to get the interaction built-in Web service interface.

Convert Web Service - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Media Go Links

Address: http://localhost/Convert.asmx

Convert

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [toFahrenheit](#)

This web service is using <http://tempuri.org/> as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. <http://tempuri.org/> is available for XML Web services that are under development, but published XML Web services should use a more permanent namespace.

Local intranet

Convert Web Service - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Media Go Links

Address: http://localhost/Convert.asmx?op=toFahrenheit

Convert

Click [here](#) for a complete list of operations.

toFahrenheit

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
pCentigrade:	22

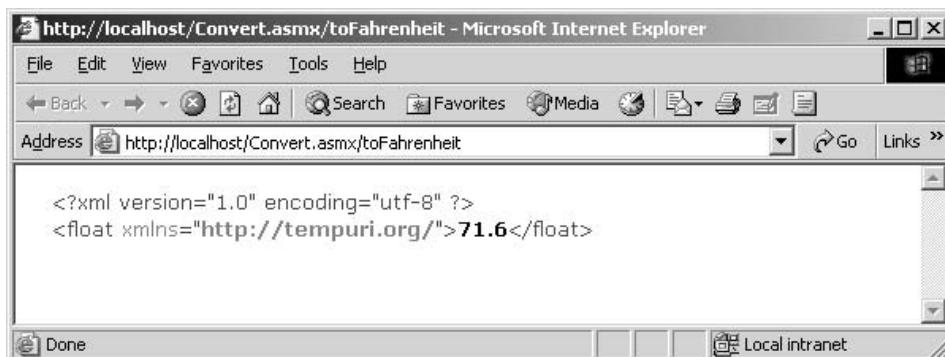
SOAP

The following is a sample SOAP request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /Convert.asmx HTTP/1.1
Host: localhost
```

Done Local intranet

The result from Web services is shown in a response window.



We can easily implement a .NET client to access this .NET Web services programmatically. We can also implement a Web services Java client accessing a .NET Web services.

8.6 SUMMARY

In this chapter, we discuss the foundation of Web services including XML, SOAP, concept and development of Web services, deployment of a Web service, publication of a Web service, and connection models of Web service.

Web service is an on-line component that can be described by its interface – WSDL, deployed by a deployment tool such as AXIS adminClient utility with WSDD, published in a registry such as UDDI, located by a SOAP lookup based on UDDI API, and invoked by a SOAP request. The Web services are loosely coupled, contracted components that communicate via XML-based WSDL Web services interface.

WSDL Web services interface plays a role of implementation of a Web services and its clients. The modification of a Web service will not affect its client, which may be another Web services or application client, as long as the contract is unchanged. A Web service is reusable. Even the part of WSDL interface definition is reusable.

A Web service component is different from other components in that Web service is an on-line component; Web service is an interoperable component; Web service is a component that can wrap other type components such as EJB, CORBA, and .NET components; Web service is a firewall-friendly component because it is XML SOAP-based text format proprietary-free component; Web service is globally available component supported by its UDDI registry; Web service is an application-to-application (A2A), business-to-business(B2B), e-commerce-to-e-commerce(E2E) component.

The connections between Web services can be synchronous and asynchronous, one-way and two-way message passing, message-based and rpc-based, message passing and request/response, and static binding and dynamic binding. A Web services composition can be a wrapper of an existing application, or containments of a number of Web services, or aggregations of a number of Web services.

Although Web services has drawn so much attention in the business industry and software development, this technology still faces a lot of challenges such as the standards for interaction and conversation between Web services and flow control in Web services integration.

8.7 SELF-REVIEW QUESTIONS

1. Web services provider and requestor
 - a. must be written in same programming language.
 - b. may be written in different languages.
 - c. may be running in different platforms.
2. An Apache Axis Web services component can be deployed in
 - a. .jws
 - b. .asmx
 - c. .wsdd
 - d. .jar
3. HTML is used for Web presentation and XML/SOAP is used for message exchange representation.
 - a. True
 - b. False
4. A Microsoft Web services component has an extension
 - a. .zip
 - b. .exe
 - c. .asmx
 - d. none of the above
5. WSDD is a utility tool used to deploy a Web services.
 - a. True
 - b. False
6. A Web services distributed component and its client must be run
 - a. in different machines
 - b. in the same machine
 - c. on Internet
 - d. all above
 - e. none of the above
7. A Java Web services component can be accessed by a .NET client.
 - a. True
 - b. False
8. WSDL is an interface between a Web services and its client.
 - a. True
 - b. False

9. UDDI is a Web services directory for Web services registration and lookup.
 - a. True
 - b. False
10. Web services request/response interaction model is a RPC-based messaging mode.
 - a. True
 - b. False
11. Three Web service design models are top-down, bottom-up, green field design.
 - a. True
 - b. False
12. Service tag in WSDL is a reusable implementation definition of a Web service.
 - a. True
 - b. False

Keys to Self-Review Questions

1. b,c 2. a 3. a 4. c 5. a 6. d 7. a 8. a 9. a 10. a 11. a 12. b

8.8 EXERCISES

1. What is a Web service?
2. How does a Web service work?
3. How does .NET support?
4. Describe the protocol stack for Web service.
5. What is the difference between CORBA and Web service?
6. List four `portTypes` of Web service.
7. Is Web service a cross-language platform?
8. How does one composite a Web service component nested inside another component?
9. Describe the difference between XML and HTML.
10. Where is a Web service component deployed?
11. Describe the difference between XML and SOAP.
12. What is WSDL?
13. What is asynchronous interaction with a Web service?
14. What is synchronous interaction with a Web service?
15. When is class channel needed?
16. What is UDDI?
17. Can a Web service provider be a client of another Web service?

18. What is the difference between MS DCOM, .NET components, and Web services component?
19. List all major Web services vendors you know.
20. Does Web services have problem with firewall?

8.9 PROGRAMMING EXERCISES

1. Design a Web services component NET C#, managed C#, .NET VB, which provides service of a simple calculator. The calculator has the functionality to perform addition, subtraction, multiplication, and division of two real numbers. It can also detect the zero divisor in division operation. Use namespace in this component.
2. Design a client of this calculator in .NET VB, .NET C#, .NET managed C#, and Java.
3. Deploy this Web services component “Calculator” as a. asmx file on IIS server.
4. Rewrite component “Calculator” as a Java Web services component.
5. Deploy it on Apache Axis by WSDD or just. jws.
6. Develop a .NET client in C# to access this Web services.
7. Display the wsdl file of the Web services.
8. Display the SOAP of the Web services request and response.

REFERENCES

- [Apache 2004a] <http://ws.apache.org>, 2004.
- [Apache 2004b] <http://ws.apache.org/axis>, 2004.
- [Deitel 2003a] Deitel. *Java Web Services for Experienced Programmers*, Prentice Hall, 2003.
- [Deitel 2003b] Deitel. *Web Services, A Technical Introduction*, Prentice Hall, 2003.
- [McGovern 2003] McGovern, James. *Java Web Services Architecture*, Morgan, 2003.

INDEX

A

Abstraction, 17
Addition, 21
Aggregation, 21, 204–205
Aggregation composition, 204
Algebra, 19
AOP, 2
Application programming interface (API), 19
Aspect-oriented programming, 2
Assembly, 26, 180–183, 198
Association, 102
Asynchronous communication, 101
Asynchronous interaction, 293–294
Asynchronous message passing, 162
Attributes, 163
AWT components, 25
AXIS (Apache extensible interaction system), 282

B

BeanBox, 47
Bean Builder, 37, 66, 73, 84
BeanDescriptor, 63
Bean development kit (BDK), 20, 37, 64, 72, 76, 78, 84
Bean icon, 54
BeanInfo class, 56–57, 62
Bean managed persistence (BMP), 92, 95
Beans, 37
Bit literals, 28
Black-box reuse, 5, 11, 267
Boolean algebra, 8, 19

Boolean literals, 27

Bundle activator class, 239
Bundle events, 251
Bundle life cycle, 247
Bundles, 239, 251

C

C#, 201–203, 224–228
Callback, 210
Canvas, 50, 52
CBSE, 10
CCM, 149, 159
CCM component, 159–160
CCM component categories, 164
CCM deployment, 183
CCM ports, 160
Chip-level components, 7
Client-side programming, 38
Cohesion, 10
COM, 195
Common Object Request Broker Architecture (CORBA), 146
 architecture, 147–149
 assembly, 180–183
 attribute, 163
 connection models, 173–176
 CORBA components, 149–158
 CORBA component model (CCM), 159
 deployment, 183–184
 deployment models, 176–184
 emit, 162

- Common Object Request Broker Architecture (CORBA) (*continued*)
 event sink, 162
 event source, 162
 facets, 161
 home interface, 163
 IDL equivalence, 161–163
 interface definition language (IDL), 147–150
 interface repository, 147–148
 module, 150
 object adapter (OA), 147–148
 object reference (OR), 148–149
 object request broker (ORB), 147
 OpenCCM, 165–173
 packaging, 176–180
 ports, 160
 publish, 162
 receptacles, 161
 skeleton, 148
 stub, 148
- Common Language Runtime (CLR), 195–196
- Common Type System (CTS), 195–196
- Component abstraction, 17
- Component algebra, 20
- Component architecture, 19
- Component assembly, 11
- Component assembly descriptor, 180
- Component associations, 25
- Component-based analysis (COA), 10
- Component-based design (COD), 10
- Component-based management (COM), 10
- Component-based software development (CBSD), 10
- Component-based software engineering (CBSE), 10
- Component charts, 20
- Component definition, 18
- Component executor, 164
- Component implementation, 11
- Component infrastructures, 18
- Component interface, 11
- Component models, 18, 20
- Component object, 11
- Component-oriented programming, 3
- Component persistent state, 164
- Component provision, 11
- Component specification, 11, 28
- Component standards, 18
- Component table, 24
- Component technology, 6, 19
- Composite components, 38
- Connection models, 18, 21
- Connectors, 19
- Conquering complexity, 4
- Container, 89–90
- Container managed persistence (CMP), 92, 95
- Containment, 204–205
- Containment composition, 204
- COP, 1, 3, 8–9
- CORBA, 146
- CORBA component, 150
- CORBA component descriptor (CCD), 178
- CORBA infrastructure, 149
- CSL (Component Specification Language), 26
- CSL semantics, 29
- Customization, 38, 40, 57
- Customizing, 57
- D**
- Data abstraction, 17
- DCOM, 195
- Decomposition, 17
- Delegate, 205
- Dependable systems, 18
- Deployment, 10, 104–106
- Deployment models, 18, 26
- Design mode, 41, 70
- Development for reuse, 11, 38–39
- Development with reuse, 11, 38–39
- Divide, 22
- Divide and conquer, 10, 17
- DLL components, 215–219
- Document type definition (DTD), 269–271
- Drag-and-drop, 37
- Dynamic binding, 9
- Dynamic connection, 247
- Dynamic discovery, 9
- Dynamic invocation, 295
- Dynamic invocation interface (DII), 148
- Dynamic link library (DLL), 196–199
- Dynamic skeleton interface (DSI), 148
- E**
- EJB architecture, 88
- EJB component, 90
- EJB component model, 90
- EJB container, 89
- EJB Web Service components, 94
- Emit, 162
- Encapsulation, 10
- Enterprise Java Bean (EJB)
 CMP example, 120–142
 component model, 90–92
 bean class, 92–93
 entity bean, 95–99
 interface, 92
 message-driven bean (MDB), 99–100
 session bean, 92–95
- connection model
 association, 102
 asynchronous, 101
 local, 101
 remote, 101
 synchronous, 101
- container, 89–90
- deployment, 104–106
- Entity Bean, 95–99, 101

Entity component, 164

Events, 20, 40

Event adaptor, 70

Event delegate, 205

Event driving, 25

Event handling, 22, 69

Event listeners, 40

Event sink, 162

Event source, 162

Event to event, 22

Event to property, 22

Export packages, 247

Export ports, 173

F

Facets, 161

Fire and event, 40

G

Gear-oriented programming, 2

Global assembly cache (GAC), 212

Graphical user interface (GUI), 37

Gray-box reuse, 5

H

Hardware components, 7

Hoare logic, 8

Home executor, 164

Home interface, 91, 95, 163

I

Identifiers, 27

Identity, 10

IDL, 147–148

Implementation repository, 148

Import packages, 247

Import ports, 173

Independent software vendor (ISV), 38

Infix operators, 28

Infrastructures, 18

Installed component, 11

Instance variables, 39

Integer literals, 27

Integrated development environment (IDE), 20

Intellectual property, 7

Interface, 10, 20, 92

Interface repository (IR), 148

Intermediate Language (IL), 195, 197–199

Introspection, 40

Invisible beans, 48

J

J2EE Architecture, 88–89

J2EE server, 89

JApplet, 54

JAR files, 44–45, 104, 240, 251

JavaBeans, 37

JavaBeans component infrastructure, 37

Bean Builder, 66

BeanDescriptor, 63

BeanInfo class, 56

component model, 38

connection model, 64

customization, 40

customizing, 57

deployment model, 72

design mode, 70

introspection, 40

invisible beans, 48

runtime mode, 70

serializable, 39, 43

persistence, 40

Java Embedded Server, 239

Java naming and directory interface (JNDI), 89

JES, 20, 239

JFrame, 47

JPanel, 41, 43, 53, 55

Just-in-time (JIT) compiler, 195

K

Keywords, 28

L

Local, 101

M

Managed C++, 203, 224–226

Managed code, 196

Managing change, 4

Manifest file, 44–45, 72, 199, 239–240

Many-sorted algebra, 29

Marshal by Reference (MBR), 207–210

Marshal by Value (MBV), 207–210

Message-Driven Bean (MDB), 92, 99–100

Message passing, 25

Methods, 20, 40

Modification of interface, 22

Multiplication, 22

Multithreading, 41

N

Name space, 197–198

.NET Component Infrastructure, 194

component deployment, 212

private deployment, 212–213

public shared deployment, 213–215, 228–230

component model, 198

C#, 201–203, 224–228

managed C++, 203, 224–226

connection model, 204–212

composition aggregation, 204–205

composition containment, 204–205

event delegate, 205

- .NET Component Infrastructure (*continued*)
 - remote asynchronous callback, 210–212
 - remote marshal by reference (MBR), 207–210
 - remote marshal by value (MBV), 207–210
 - framework, 194–237
 - common language runtime (CLR), 195–196
 - common type system (CTS), 195–196
 - dynamic link library (DLL), 196–199
 - intermediate language (IL), 195
 - .NET Web Service, 304–306, 309–311
- O**
 - Object adapter (OA), 147–148
 - Object-oriented programming, 2
 - Object reference (OR), 148–149
 - Object request broker (ORB), 147
 - OOP, 2, 8–9
 - Open CCM, 165–173
 - ORB, 147
 - OSGi components, 238
 - OSGi component infrastructure, 20, 262
 - bundles, 239
 - bundle events, 251
 - component model, 239
 - connection model, 247
 - deployment model, 250
 - export packages, 247
 - import packages, 247
 - service dependency, 250
 - service events, 251
 - OSGi framework, 239
 - OSGi implementations, 239
 - Open Service Gateway initiative, 238
 - OSGi specification, 238–239
- P**
 - Packaging, 115, 176–180
 - Package dependency, 248
 - Passing by reference, 92, 101
 - Passing by value, 92, 101
 - Persistence, 38, 40, 43
 - Platform-independent components, 38–39
 - Plug-and-play, 18, 239
 - Ports, 160
 - Post-condition, 21
 - Pre-condition, 21
 - Prefix operators, 28
 - Private component, 199, 212
 - Procedural abstraction, 17
 - Procedures, 8
 - Procedure-oriented programming, 2
 - Process component, 164
 - Programming, 1–3
 - Programming elements, 8
 - Properties, 20, 40
 - Property customization, 68
 - Property descriptor, 63
 - Property file descriptor, 178
- Provided interface, 11
- Proxy component, 68
- Public component, 199, 212
- Publish and emit, 162
- R**
 - Receptacles, 161
 - Reduction, 21
 - Reflection, 38
 - Remote, 101
 - Remote interface, 91, 95
 - Remote asynchronous callback, 210–212
 - Remote connector, 207–210
 - Remote method invocation (RMI), 100
 - Remote procedure call (RPC), 147
 - Required interface, 11
 - Reusability, 17
 - Reuse, 5
 - Runtime mode, 41, 70
- S**
 - Security, 41
 - Serializable, 39, 43
 - Serialization, 38–39
 - Server-side programming, 38
 - Service component, 164
 - Service dependency, 250
 - Service events, 251
 - Service interface, 246
 - Session bean, 92–95, 125
 - Session component, 164
 - Self-contained, 6, 12
 - Self-deployable, 6, 12
 - Simple object access protocol (SOAP), 195, 271
 - Skeleton, 148
 - Stateful session bean, 94
 - SOAP messages, 271
 - Software component, 7
 - Software component definitions, 6
 - Software dependability, 17
 - Software development process, 11
 - Software package descriptor, 178
 - Software productivity, 18
 - Software standardization, 18
 - Sort, 29
 - Source component, 20, 25
 - SQL access, 92
 - State operators, 28
 - Statement-level programming, 8
 - Static connection, 247
 - Static invocation, 295
 - Static invocation interface (SII), 148
 - Static skeleton interface (SSI), 148
 - Structured programming, 9
 - Stub, 148
 - Subroutines, 8
 - Subtraction, 21
 - Swing components, 25

Switch-oriented programming, 2
Synchronous, 101
Synchronous connection, 101
Synchronous interaction, 293, 295

T

Target component, 20, 25
Temporal logic, 24
Temporal operators, 24, 28
Third-party composition, 9

U

UDDI (Universal Description, Discovery and Integration), 26, 272, 275
Unmanaged code, 196

V

Visibility, 40
Visual Studio .NET, 215–224
 DLL components, 215–219
 window form client, 219–221
 Web form client (VB.NET, ASP.NET), 221–224

W

Web services
 Apache axis, 282, 286, 299–309
 architecture, 272–280
 asynchronous, 293–294
 composition, 295–297

connection model, 293–297
component model, 92, 281
development, 282–293
deployment, 297–299
deployment descriptor, 297
document, 269–271
Document Type Definition (DTD), 269–271
dynamic, 295
framework, 267–269
synchronous, 293–297
SOAP, 271–272, 302–304
static, 295
.NET Web Service, 304–306, 309–311
wrappers, 296
WSDD, 297–299
WSDL, 281–293
XML, 269–271
White-box reuse, 5
Window form client, 219–221
WSDL, 91, 273, 281–293
WSFL, 296

X

XML, 269–271
XML-based deployment, 26
XML descriptor, 176
XML file, 66, 71, 73, 104, 181
XML schema, 269
XML schema document (XSD), 270