

Integration-Ready Architecture and Design

Software Engineering with XML, Java, .NET, Wireless,
Cryptography and Knowledge Technologies



Jeff Zhuk

CAMBRIDGE

CAMBRIDGE

www.cambridge.org/9780521525831

This page intentionally left blank

INTEGRATION-READY ARCHITECTURE AND DESIGN

What would you do if your IT job was no longer performed in your country? Your survival does not lie in limiting global collaborative engineering. IT workers will survive and prosper because of their ability to innovate, to quickly learn and change directions, and to evolve from Information Technology into Distributed Knowledge Technology. You have no choice but to be pro-active, learn to stay current, and even run ahead of the game.

Integration-Ready Architecture and Design shows how to build presentation factories and seamless integration of VoiceXML, WAP, and Web technologies, providing access to corporate data and services not only through PCs and corporate workstations, but also through multiple types of wired and wireless devices and PDAs. The author integrates theory and practice, going from foundations and concepts to specific applications and architectures. Through deep insights into almost all areas of modern CIS and IT, he provides an entry into the new world of integrated knowledge and software engineering. Readers will learn the “what’s, why’s, and how’s” on: J2EE, J2ME, .NET, JSAPI, JMS, JMF, SALT, VoiceXML, WAP, 802.11, CDNA, GPRS, CycL, XML, and multiple XML-based technologies including RDF, DAML, SOAP, UDDI, and WDSL.

For Internet and wireless service developers, this book contains unique recipes for creating “integration-ready” components. Architects, designers, coders, and even management will benefit from innovative ideas and detailed examples for building multi-dimensional worlds of enterprise applications. Throughout, the book provides a “unified service” approach while creating a core of business frameworks and building applications for the distributed knowledge marketplace.

Jeff Zhuk is the President of Internet Technology School. A software architect and developer with more than twenty years of experience and numerous patents and publications, he teaches at the University of Phoenix and DeVry University, and he conducts corporate consulting and training. He has pioneered IPServe.com and JavaSchool.com, which promote collaborative engineering and a distributed knowledge marketplace. An expert in distributed enterprise applications and wireless, XML, and Java technologies, his current focus is on the integration of software and knowledge engineering in a new development paradigm.

Look out for code examples and updates on the book’s Web site www.cup.org/Titles/0521525837.htm

This book is dedicated to my parents, Lubov and
Veniamin, and to my wife, Bronia.

INTEGRATION-READY ARCHITECTURE AND DESIGN

SOFTWARE ENGINEERING WITH XML, JAVA, .NET, WIRELESS, SPEECH, AND KNOWLEDGE TECHNOLOGIES

JEFF ZHUK

Internet Technology School, Inc.



CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS

Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press

The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org

Information on this title: www.cambridge.org/9780521525831

© Jeff Zhuk 2004

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2004

ISBN-13 978-0-511-13378-7 eBook (Adobe Reader)

ISBN-10 0-511-13378-2 eBook (Adobe Reader)

ISBN-13 978-0-521-52583-1 paperback

ISBN-10 0-521-52583-7 paperback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

Preface	<i>page xi</i>
Contributors	xv
Acknowledgments	xvi
Introduction	xviii
Notes for Educators: AMA Teaching Methods	xxix
Chapter 1	
Collaborative Engineering	1
Management Style as an Important Part of the Development Process: True Leaders Versus “Pure” Managers	2
Development Methodologies: Capability Maturity Model and More	4
Extreme Programming: Rules of the Game	5
Six Sigma	5
Distributed Collaborative Development	5
24×7 Distributed Development Practices	8
Steps in the Process	9
Basic Steps of the Development Process with an Object-Oriented Approach	10
Learn by Example: Compare OOP and Procedural Programming	21
UML Notations	24
Example of Object-Oriented Analysis: Create an OMD for Document Services	26
Create the DocumentService Model	26
Architecture Steps: Find Playground-Tiers for Your Objects	27
From Single-User to Client-Server and Multi-Tier Architecture Models	28
Basic Design Steps and Rules	33
Instead of a Summary: How Direct Access to Products and Services Improves the Balance of Supply and Demand	40

Chapter 2	
Software Architecture and Integration Technologies	42
Software Architecture—The Center Place of Software Development	42
Architectural Elements, Styles, Views, and Languages	43
Programming Styles	48
Integration Technologies	49
Object Linking and Embedding (OLE) and ActiveX	49
CORBA and IDL	51
Microsoft's Distributed COM Architecture	52
Java Technology Architecture	53
Java Applet	53
The Java Bean Architecture	53
J2EE Architecture	55
Java Server Pages	55
The Enterprise Java Beans Architecture	56
EJB Components and Containers	56
The Java RMI Architecture	57
The Bridge from Java to CORBA Technology	57
XML Web Services	59
An Example of an XML-Based Service API	60
Additional Benefits: Ability to Add or Change Services at Run-Time	63
How Can We Register a New Web Service?	64
Is There a Mechanism to Pack Both Data and Services into a Message?	64
How Do Software Vendors Support Web Services?	64
 Chapter 3	
From a Specific Task to “Integration-Ready” Components	69
User Requirements, Version 1: The “News Watch” Applet	70
User Requirements, Version 2: The Reusable <i>NewsLine</i> Component	79
User Requirements, Version 3: View Multiple Web Information Channels in the <i>NewsLine</i> Component	88
Integration-Ready Service Components and Extensible Service Containers	100
An XML-Based Configuration File	103
Start from the Parameters for a Single Component	105
How Would We Use XML-Based Parameters while Building Components?	106
Event-Handling Procedure	114
What Is the <i>Dispatch()</i> Method for and How Do We Define Its Function?	115
Provide the Possibility of Interactive Components (Event Handling)	125
The <i>ControlComponent</i> Class Description	127
Reuse, Not Abuse	130

Chapter 4	
Integration with Voice	133
What Is the Base for Creating a Voice Component?	133
How Are Voice Components Coded?	140
Chapter 5	
An Introduction to Knowledge Technologies	151
Ontology	152
DAML+OIL: A Semantic Markup Language for Web Resources	153
Topic Maps	156
Data-Mining Process and Methods	160
Frames and Slots	161
The CycL Language	163
How to Begin with OpenCyc	173
Chapter 6	
Write Once	194
Multiple Types of Data Storage	195
Control Systems and Controllers	199
Document-Handling Services	203
Chapter 7	
The New Generation of Client–Server Software	222
What Are the Best Ways to Provide Client Functionality?	225
Thin Clients	226
Multiple Scenarios	235
Synchronous or Asynchronous Client-Server Communication	243
Example of XML Multiple-Screen Scenario	244
Good Performance Follows Good Design	251
Keep a Stable API while Changing Communication Mechanisms	252
Add Open Office Features to Rich Clients	254
How Much Abstraction Is Too Much?	255
Chapter 8	
Wireless Technologies	257
Wireless TDMA, CDMA, GSM, and Other Standards	258
802.11 and WLANs	261
Bluetooth Technology	262

SMS: The Most Popular Wireless Service	262
What Is WAP?	263

Chapter 9

Programming Wireless Application Protocol Applications **267**

Rethink the Existing Web Page Paradigm in WAP Terms	268
What Is a Presentation Factory and How Do You Create One?	269
Programming WAP/WML Pages	270
Secure Transactions with WML Factory	271
Is the Data Size Too Big for a Device? Not a Problem!	279
WAP Push	281
WAP Devices and Web Services	284

Chapter 10

A Single JavaCard Identity Key for All Doors and Services **286**

What Is a Smart Card?	286
What Is a JavaCard?	287
Why Do We Use Multiple Keys?	287
Can We Have a Single “Identity Key”?	288
How to Program JavaCards	288
What Are JavaCard Programming’s Limitations?	290
The <i>javacard.framework</i> Package for JavaCard Programming	290
Writing a Sample JavaCard Applet: <i>VirtualCurrency</i>	291

Chapter 11

The J2ME Family **306**

The MIDP	308
How Do We Display Screens and Handle Events?	308
Multimedia on Wireless	309
What Are MIDlets?	310
Can We Store Data on Mobile Devices?	310
<i>MIDlet</i> Security	311
Can We <i>Push</i> from a Server to the MIDP Device?	311
Wireless Messaging with Short Message Service and Other Protocols: An Important Component in Our Application	312
Wireless Messaging Client Application	314

Chapter 12

Speech Technologies on the Way to a Natural User Interface **338**

What Is a Natural User Interface?	338
Speaking with Style	339
Java Speech API Markup Language	341

Speech Recognition with Java	348
Microsoft Speech SDK	351
Speech Technology to Decrease Network Bandwidth	354
Standards for Scenarios for Speech Applications	359
Speech Application Language Tags	360
Grammar Definition	361
VoiceXML	364
ECMAScript	378
Grammar Rules	380
The VoiceXML Interpreter Evaluates Its Own Performance	382

Chapter 13

Integration with Knowledge

387

Why Are Computers Stupid? What Is Missing?	387
Knowledge Integration Participants, Processes, and Products	388
Connect Software and Knowledge Technologies	390
What Are the Main Goals of the Knowledge Connector Package?	392
Object Model Diagram	393
Formatting and Presentation Layers	393
The Magic of Service Invocation	394
What Does It Mean to Play a Scenario?	415
Do You Want to Be a Scenario Writer?	416
Application Scenario Language	416
Installing and Running the Package	438

Chapter 14

Distributed Life in the JXTA and Jini Communities

446

Distributed Processing and the Flat World of XML	446
What Is JXTA?	449
Jini	463
JXTA and Jini: Just Neighbors or Collaborators?	466

Appendix 1

Java and C#: A Saga of Siblings

468

Java Virtual Machine and Common Language Run-Time	468
From Basics to the Next Level on the Java/C# Programming Trail	484

Appendix 2

XML and Web Services

539

XML Extends the Web and Builds a Playground for Its Children	539
XML Describes Business Rules and Data Structures; XSLT and X Path Describe Their Transformations	539

XML Provides Direct Hooks to Services on the Web with SOAP, WSDL, and UDDI	540
Interactive Web with XForms	540
XML in Voice Applications	540
XML Drives Semantic Web and Knowledge Technologies	541
XML Web Services	541
Web Services at Work	541
Encode Service Requests with SOAP	542
Describe Web Services with WSDL	542
Publish and Discover Web Services with UDDI	544
Business Process Execution Language for Web Services	544

Appendix 3

Source Examples

551

Getting into Collaborative Services	551
The MIME Multipart/Related Content-Type; RFC 1867	554
Working with Geographical Information Systems	570
Reading AutoCAD Vector Graphics	577
JNDI: What, Why, and How?	578
Instant Screen Share	581
Instant Voice Share with JMF	581
Java Messaging Services (JMS): A New Way to Think about Applications	587
Create Speech Recognition and TTS Applications in C#	589
Fight Email Spam and Increase Email Server Efficiency	590

Index

599

Preface

WHO SHOULD READ THIS BOOK?

Integration-Ready Architecture and Design strives for a union of theory and practice. Teaching the latest wired and wireless software technologies, the book is probably the first entry into “the next big thing,” a new world of integrated knowledge and software engineering. Written by a software architect and experienced trainer, this book is for:

- Software architects, designers, and developers
- Internet and wireless service providers
- IT managers and other IT professionals, as well as amateurs
- Subject matter experts who will directly participate in a new development process of integrated software and knowledge engineering
- Students and educators, who will find up-to-date materials for the following courses:
 1. Software Architecture,
 2. Software Engineering,
 3. Programming Concepts,
 4. Information Technologies,
 5. Smart Card and JavaCard Technologies,
 6. Wireless Technologies,
 7. J2ME and Wireless Messaging,
 8. XML Technologies,
 9. Speech Technologies,
 10. Java Language and Technology,
 11. C# and .Net Technology,
 12. Integration Technologies,
 13. Business Communications and Collaborative Engineering,
 14. Web Technologies,
 15. Introduction of Ontology, and
 16. Integrated Software and Knowledge Engineering (introduced in the book)

- **Peers: students, instructors, consultants, and corporate team players** who might start using a peer-to-peer educational tool offered in the book as their entrance to the distributed knowledge marketplace
- **All of the above who want to know** how things work, should work, and *will* work in the IT world

WHY DID I WRITE THIS BOOK?

Of course, I wanted to solve several global-scale problems. Divided by corporate barriers and working under “time-to-market” pressure, we often replicate data and services and produce software that is neither *soft* nor *friendly*. Working as fast as possible in this mode, we deliver products that lack flexibility and teamwork skill and are hardly ready for integration into new environments. These products strictly target user requirements—which become obsolete by the time the project ends. Producing “more of the same” and raising the number of product choices (instead of moving to new horizons), we actually increase entropy and slow down the progress of technology, which depends heavily on inventions, new usage, or new combinations of existing tools and methods.

The famous formula “write once” is not working *anywhere* today. One of the reasons is the absence of a mechanism capable of accepting, classifying, and providing meaningful information about new data or services created by knowledge producers.

We have not changed our way of writing software during the past twenty years.

We have not moved far from the UNIX operational environment (which was a big hit thirty years ago).

Our computers are much faster, but for the regular user, they are as stupid as they were forty years ago. We add power but we fail to add common sense to computers, we cannot help them learn, and we routinely lose professional knowledge gained by millions of knowledge workers.

Meanwhile, best practices in software and knowledge engineering are reaching the point of critical mass. By learning, understanding, and *integrating* them, we can turn things around. We might be able to improve the reliability of quickly changing environments by using distributed self-healing networks and knowledge base–powered application solutions.

We can finally stop rewriting traditional address book, scheduling, inventory, and order applications. We will shift our focus from ironing out all possible business cases in our design and code to creating flexible application mechanisms that allow us to change and introduce new business rules on the fly. Coming changes are similar to the transition from structural to object-oriented programming. We are going back to school.

HOW TO USE THIS BOOK

I hope you find this book on your list of recommended reading or as “the best gift for yourself, your spouse, and your friends.” Just buy two copies and let *them* figure out what to do with the book. In the worst case, it can be used for self-defense. It is almost as heavy as other good books.

If your gift list includes yourself, you might want to read this book in the bookstore first—at least some selected chapters, starting from the back.

For example, Chapter 10, about a JavaCard key that opens all doors, can be very handy the next time you lock your keys in your car. If this happens too often to you or your close relatives, you might find Chapter 11, on J2ME and wireless messaging, very practical.

Armed with the knowledge of wireless technologies from Chapters 8, 9, and 11, you can create your own communication service and finally stop switching from AT&T to Sprint and back to Verizon. Serve your friends and neighbors, compete with T-Mobile, and someday I'll be happy to buy your integrated "wireless portal communicator" product.

If you are a serious developer or plan to become one, you might prefer to start from the beginning and read all the way through. Search for long-term, secure, and exciting IT directions. Find out why all the pieces of the puzzle, as well as the glue, are almost equally important. Teach yourself to see every technology (component) as an object with three dimensions: what, why, and how. After reading the book, you might even become less serious and more efficient.

If you want to increase your business clientele from the 20% of the population who are fluent in current computer interfaces to the rest of us, including those who hate computers or cannot bear their stupidity—just go for it! Read Chapters 4, 5, 12, and 13, on speech and knowledge technologies, and create a natural user interface, a bridge from your business to humankind.

A professional hacker (whose average age is 15 but ranges from 6 to 66) might start with Appendix 3 ("Source Examples"). Find examples that can help to build collaborative and location-based services, screen/voice instant sharing and security monitoring, and speech and distributed knowledge alliance applications. Look there for spam killer hints to be ahead of the game.

If you just want to speak more languages, go to Appendix 1 ("Java and C#: The Saga of Siblings"). You can get two for the price of one, including the latest JDK1.5 language innovations.

If you would like to include XML in your repertoire, add Appendix 2 ("XML and Web Services"), which covers several dialects of the XML family.

Chapters 5 and 13 are not only for computer folks. The elusive category of "knowledge workers"—anyone who has gained knowledge and never had a chance to share—might be looking at the Promised Land. Subject matter experts (SME)—who used to talk to developers about *what* and *why*—can find in those chapters new ways to say *how*.

There is also a downloadable software product with this book. Students and educators can use the tool for collaborative work in team projects. The tool helps to connect students and instructors with educational knowledge resources. This can elevate the visibility and quality of student projects and transmit the best of them into industry contributions. The software can be handy in academic/corporate alliances.

WHAT IS THIS BOOK ABOUT?

- The what's, why's, and how's on: J2EE, J2ME, .NET, JSAPI, JMS, JXTA, JMF, SALT, VoiceXML, WAP, 802.11, CDMA, GPRS, CycL, XML, and multiple XML-based technologies, including RDF, DAML, SOAP, WSDL, and UDDI. The book turns these abbreviations into understandable concepts and examples
- The distributed knowledge marketplace
- Collaborative engineering methods and technology

- XML and Web technology architecture, design, and code patterns
- Ontological (knowledge) engineering and natural user interface
- XML-based application scenarios that integrate dynamic user interface and traditional services with speech and knowledge technologies
- Unique recipes for creating integration-ready components across a wide range of client-server, multi-tier, and peer-to-peer distributed architectures for Internet and wireless service developers
- Innovative ideas, methods, and examples for building multidimensional worlds of enterprise applications
- Privilege-based access to corporate data and services, not only through PCs and workstations but also through multiple types of wired and wireless devices and personal digital assistants with seamless integration of wireless, Web, speech, and knowledge technologies
- A unified approach to architecture and design that allows for J2EE and .NET implementations with code examples in Java (most of the source code) and C# languages
- Integration of software and knowledge engineering in knowledge-driven architecture
- Union of theory and practice. As many of us do, I divide my professional time between the education (30%) and development (70%) worlds. Time sharing is not always the best option in real estate, but I hope it works for this book by providing more cross-connections between these parallel worlds. Like most parallel worlds, they have (almost) no intersections according to Euclidian geometry, but we can find many by moving into Riemann's space
- Questions and exercises, case study assignments, design and code samples, and even a learn-by-example self-training system that allows you to enter the distributed knowledge marketplace

Bridging the gap for a new generation of software applications, the book teaches a set of skills that are becoming extremely valuable today and that will certainly be in high demand tomorrow.

This book is designed to walk a reader through the peaks of current software, with a focus on foundations, concepts, specifications, and architecture. The hike then goes down to the valley of implementation, with detailed design examples and explanations, and finally flies to new horizons of software and knowledge technologies. There lies the happy ending, where software and knowledge engineering ride off into the sunset together.

Contributors

Many special thanks to the people who made direct contributions to this text:

Ben Zhuk—Cowrote a number of sections, edited the entire book for both content and style, and created all diagrams and illustrations (except those mentioned below). He was a sounding board for ideas throughout the writing process and was an invaluable resource. I am indebted to him, more than I can express, for his tireless efforts and the countless hours he put into this project.

Dmitry Semenov—Senior Software Architect who read the manuscript carefully and thoroughly. Dmitry's remarks and criticism helped me to clarify content and add significant parts to Chapters 3 and 13.

Olga Kaydanov, artist—Provided great design ideas and artistic inspiration for illustrations (<http://artistandart.com>).

Irina Zadov, artist—Illustrated Fig. 1.1, Fig. 1.2, Fig. 1.5, and Fig. 1.4 (<http://ucsu.colorado.edu/~zadovi>).

Inna Vaisberg, designer—Illustrated Fig. 3.1 and Fig. A3.18 (<http://javaschool.com/skills/Inna.Vaisberg.Portfolio.pdf>).

My former students, talented software developers:

Masha Tishkov, who wrote and tested several XML parser methods in the `Stringer.java` source and helped to prepare the uploader package;

Slava Minukhin, who teamed up with me to write C# sources for Appendix 3: `TextToSpeech.cs`, `SocketTTS.cs`, `Recognizer.cs`;

Alex Krevenya, who helped write email- and spam-related sources for Appendix 3; and **Dina Malkina**, who wrote C++ sources for Chapter 12: `ListeningClient.cpp` and `TalkingClient.cpp`.

Thank you so much!

Acknowledgments

This is a great opportunity to say thank you to so many people without whom this book would be impossible. Thanks to my parents; if not for them, you might be holding a different book right now.

To my friends (many of them former students), who assisted and supported me with many essential steps in the book's production.

To Candi Hoxworth, IT Manager, who read most of the book and provided great suggestions on improving my American accent in it.

To Stuart Ambler, Mathematician and Software Architect, who reviewed and provided important feedback for Chapters 8, 10, and 12.

To Michael Merkulovich, Software Team Lead, who read and provided valuable suggestions for Chapter 6 and Appendix 1.

To Nina Zadov, Senior Software Engineer, who read and approved Chapter 7.

To Roman Zadov, Mathematician, who reviewed Chapter 5 (Ontology).

To Bryan Basham, Java Instructor, who reviewed a section (multipart/form upload) from Appendix 3.

To Linda Koepsell, Course Development Project Manager, who reviewed a section from Chapter 11 (J2ME).

To Jason Fish, Enterprise Learning, President, who invited me to teach for Java University at an international conference, where I met Lothlorien Homet from Cambridge University Press. Thank you Jason and Lothlorien, you got me started with this book.

To Cambridge University Press editors Lauren Cowles and Katherine Hew and TechBooks project manager Amanda Albert and copy editor Georgetta Kozlovski for their work and dedication.

To Cyc Corporation knowledge experts John De Oliveira, Steven Reed, and Dr. Doug Lenat, who taught me ontology and the Cyc Language and reviewed several sections from Chapters 5 and 13.

To my colleagues from the University of Phoenix, Mary A. Martin, Ph.D., Blair Smith, Stephen Trask, Adam Honea, Ph.D., and Carla Kuhlman, Ph.D., who reviewed sections from Chapter 2 (Software Architecture) and Notes for Educators.

To my colleagues from DeVry University, Ash Mahajan, Karl Zhang, Ph.D., and Mike Wasson, who reviewed the Preface and Introduction.

To Victor Kaptelinin, Ph.D., Umea University (Sweden), professor with whom I discussed multiple ideas for collaborative environments.

To Jay DiGiovanni, Director of Software Development, who reviewed and provided encouraging remarks on a section from Chapter 13.

To Vladimir Safonov, Ph.D., St. Petersburg University, Professor, who offered excellent suggestions for Chapter 2 (Software Architecture).

To Vlad Genin, Ph.D., Stanford University and University of Phoenix, Professor of Engineering, who gave me important notes on introductory sections.

To Robert Gathers, GN President, with whom I discussed the future of distributed networks and who reviewed several sections from Chapters 13 and 14.

To Rachel Levy, whose reviews of and ideas for the Introduction and Preface were inspired and right on-target.

To my children, Julie and Ben, for their moral support and phenomenal help during the entire process.

And finally, and most importantly, to my wife, Bronia, who makes everything in my life possible.

Introduction

One might think that the software industry is performing very well because it is armed with object-oriented approaches, Web services, Java and .NET technologies, and so forth. Unfortunately, this is not true.

There may be something wrong with the way we write programs. The process has not changed much during the past twenty years, except that applications and tools are getting bigger. Yet are they better and more scalable? Do they require any common sense? Can they be reused in different circumstances?

If these things were true, I do not think we would be rewriting the address book, schedule, order, and inventory applications over and over again instead of moving to new, untouched tasks. We would be able to accumulate the professional knowledge gained by millions of knowledge workers (everyone who manages information flow on a daily basis) instead of routinely losing it, as we do today. We would also not be facing the current IT crisis.

We could even have had more precise and direct access to the market's supply and demand, which would have reduced the glaring inefficiencies of the software marketplace of the 1990s. A big change is required to return investors' confidence to IT, and, hopefully, the change is coming.

Yes, technology can help economic stability if applied with precision. Sometimes I wonder why big companies are constantly growing bigger while small ones tend to disappear. Why do corporations prefer doing business with a few vendors, or often a single vendor, even when it is an expensive one? One of the reasons is that the integration of multiple vendors' products would be even more expensive.

WHY IS PRODUCT INTEGRATION SUCH A PAIN?

Products and services are currently designed to cover *predefined* tasks and work with *known* data.

Are they ready for *unknown* tasks and new data sources?

Are they ready for integration with other products and services?

Are they ready to be extended into the wireless world?



FIGURE I.1. Cooking Applications on Demand.

We think we know what is going on in our industry, but how close are we, really, to even asking the right questions?

Where can you look for information? Google.com is one answer. Where can you find any tangible product to buy? Ebay.com is a good solution. Can you find a service you need? This question is a bit harder, and the short answer is no, except for the technologies to register *Web services*, which we explore in following chapters. The ocean of industries and markets is filled with a myriad of services, but we notice only the brightest and loudest fish on the surface and have no equipment to help us swim the depths and explore this underwater universe.

If we somehow find a service, then we have a chance to take a closer look. We may subsequently find that it is not exactly what we *really* want.

Can we modify a product or service? Can we easily plug it into another service? Can I cook my application as easily as my lunch, from products and services selected right now for a single usage, as in Fig. I.1?

What does it mean to build *integration-ready* products?

Why and how do we use XML, Java, .NET, wireless, and speech technology components?

What is the next big thing? Is it about collaborative engineering, distributed knowledge marketplace, *knowledge-driven architecture*, or all of the above?

This book answers these questions, walks readers through the edge of current technologies, explains their fundamentals and interdependencies, helps to integrate the best practices of existing technologies into a new development paradigm, and provides an entrance into the world of knowledge alliances.

Examples, case studies, and the self-training application offered by this textbook arm students and instructors with ammunition for successful teamwork.



FIGURE I.2. A Mysterious Planet.

AN ILLUSTRATIVE PASSAGE FROM “BEN’S REAL AND DREAM MEMOIRS”

A hint: even if Ben were a real person (I doubt it) his stories are not necessarily real.

“If you are not a lucky person, you’d better get some skills.” I suddenly appreciated this aphorism praising the advancement of knowledge as I rushed down the runway. I had just realized that, having failed to check the schedule, I was about to miss the last kicker-ship and now risked spending the night on the tiny planet known only as H.235.

This was strictly forbidden by travel rules. H.235 was one of the infamous “black holes” rarely visited by tourists. There were rumors about strange forms of nightlife on the planet. No, this was not related to sex, but to unions of magicians, the legendary “global mind,” and disappearing travelers. I tried to flag down a couple of Hsters, but they did not understand English.

They immediately threw many extremely strange and incomprehensible words at me and then moved on when my face showed no understanding. I had always suspected that the people here spoke a different language, but I never expected it to go this far.

The street was now empty except for an old man coming my way. My memory struggled to find the right words, and a single phrase came out: "Sprechen sie Deutsch?" This was the sole gem in my collection of foreign language knowledge.

The old man stopped and smiled while taking out his cell phone. My first thought was, "He is calling the police!" Then the second, "He is calling a gang!" The man spoke the strange language into the phone, repeating a word that sounded almost, but not quite, completely unlike the word Instructions. Suddenly, he handed the phone to me.

A voice from the phone spoke clear English. "This is the Common Brain help line. You are granted a guest level of permission to ask any question, except those related to H.235 security. Your question will be translated into our language. Please end your question with the words exit and translate."

I started asking rapid-fire questions. The phone worked like magic.

After the first question, the voice from the phone took a leading role. It navigated me through a question-and-answer session, helping me express my needs in system-specific terms. At the end of the session, the phone commanded: "Mister Ben, please pass the phone to the answering party."

The old man, whose name turned out to be Paul, played the role of the answering party. He listened to the phone's version of my queries and made strange sounds that came to me via the phone in perfect English. At the end of the call, Paul used his phone as a camera to photograph my smiling face.

The voice from the phone explained that I had missed the regular night shuttle, but could still make the special shuttle reserved for just such occasions. My picture was in the system now to allow me past the auto flight attendant. I was provided with precise directions over the phone, and at one point, a map was even displayed on the phone's screen.

I felt so happy I even tried to ask a couple of extra questions, but the only answer I received was "Questions about religion or sex cannot be answered, as they are related to H.235 security."

While on the way home in the shuttle, I thought about the Common Brain. Was it a high-tech company or a psychic organization? Or maybe both would be needed to help people understand each other?

Where was the catch? What technology was used? Could the old man have been a magician? I had heard the rumor that some magicians had access to a legendary "global mind." Why was there a special flight just for me? And why was this flight taking so long?

My eyes closed as a white fog slowly surrounded me.

BACK TO REALITY: MOSAICS OF TECHNOLOGIES

That story sounds like science fiction, except for the final questions. These questions bring us back to reality, introducing *mosaics of technologies*.

Let us answer one question at a time. Was it technology or magic? Figure I.3 illustrates three related technologies that could be used today to make that science fiction story a reality.

Web technology is a respected parent: young, but mature. It has already raised a couple of very promising kids, and more are expected. A Web client or Hypertext Markup Language (HTML) browser talks directly to a Web server using HTML over the Hypertext Transfer Protocol (HTTP). HTML pages deliver text, graphics, and sound files over wired networks.

One of the Web's children is based on the wireless application protocol (WAP) and wireless markup language (WML). The small screen of a cellular phone serves as a destination for

Wireless and Speech Technologies

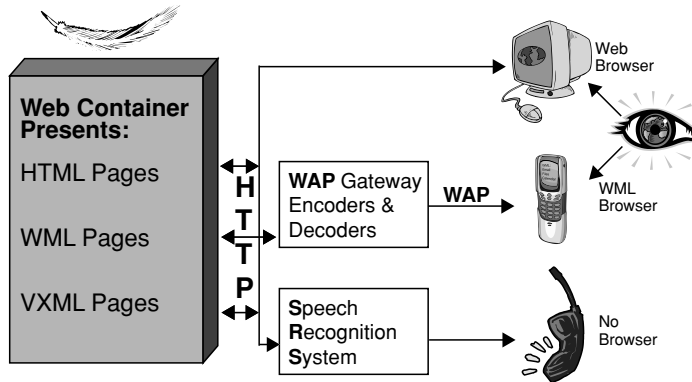


FIGURE I.3. Children of Web Technology.

WML pages. Not all wireless phones are capable of displaying WML pages; the phone must include a WML browser that provides this capability.

The other child of Web technology is based on a speech recognition system (SRS) and Voice Extensible Markup Language (VoiceXML or VXML) to deliver content to human beings in the most natural way possible. In all cases, the content is kept on the Web server or is generated dynamically by a Web container that includes a Web server and the software responsible for dynamic content generation.

In every case the content is presented by some kind of XML page, as XML is a common denominator for HTML, WML, and VXML. (See <http://w3.org> for a further description of this family.) SRS/VXML can be used for language translation. WAP/WML can be used for electronic document exchange: the photo ID that was created on the spot was sent to the server, and a map with directions was sent from the server back to the phone.

FROM CENTRALIZED COLLECTIONS TO INTEGRATION-READY SERVICES

Here is an interesting puzzle stated in a simple question. It looks like we are moving more and more resources to the server side. Is this the way technology is going? Will we face a huge service repository that continues to grow and collect more and more services?

Not necessarily. Another answer to our question is to improve the capacity for integration. This means that a service (in the simplest case, an application) might be made flexible enough to allow it easily to adapt to novel contexts. A library of services can be integrated on the fly and used on demand in real time and in a fashion transparent to the end user. For example, a wireless communication channel service, a map service, a shuttle-scheduling service, and a personal security authentication service might configure themselves to provide a new, integrated service to the end user, such as booking a flight.

This solution is more than a theoretical possibility. Businesses are already beginning to turn the focus of development from “more products in a package” to “friendly (easy to integrate and customize) products.” Figure I.4 represents a multidimensional view of integration-ready services.

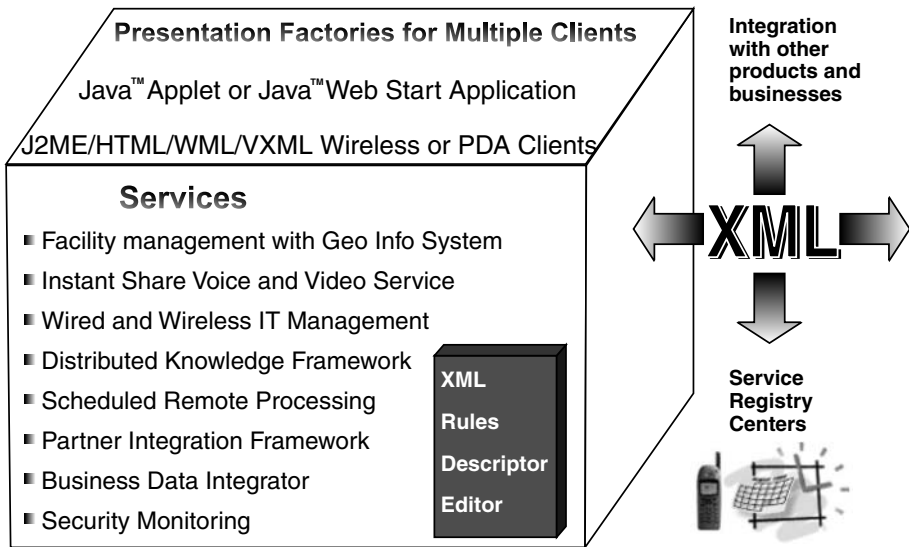


FIGURE I.4. Multidimensional Views of Enterprise Applications.

Perhaps you have heard about the “80/20” rule, which states that every service is focused on just the 20 percent of the possible scope that will produce 80 percent of the desired results. This approach becomes even more beneficial when architecture, design, and development are done in such a manner that customization is an inexpensive process, when it is even possible to add or customize services without changing the core code.

What is important for corporate businesses that are selecting the right technologies?

System flexibility, the capacity to recapture business rules and structures, and readiness for integration with other products and systems are all rapidly becoming priorities. How can these goals be accomplished?

There are several methods that can be used for system integration. Java-specific integration techniques, such as JAIN (Java application program interfaces [APIs] for integrated network), introduce system integration. There is also a new world of Java-based Jini devices capable of promoting or announcing their skills/capabilities over the network. We consider these technologies and talk about a *unified service and data* integration approach.

This XML-based approach to integration interfaces defines system behavior as well as the data integration process. It paves the road for creating integration-ready services, defines unified integration protocols and policies, and helps to connect them into distributed business alliances.

Business clients access corporate repositories with multiple types of wired and wireless devices and personal digital assistants (PDAs). Corporate workstations still outnumber other devices, but this is changing rapidly.

We do not want to redevelop business services for every type of access. It is important to understand that we need only to build more ways to present/access the same content of enterprise data and services. This leads to one of the most important architecture rules: Separate business from presentation logics. This separation is very visible in Fig. I.4.

Presentation factories (software responsible for visual/audio depictions or presentations for different types of clients) allow us to reuse a set of framework services (e.g., facility management) **and** deliver service results properly formatted for multiple types of devices from corporate workstations to wireless phones or PDAs.

In the preceding example, common business problems are solved with several “core” frameworks. Each service was focused on just 20 percent of the possible scope that produced 80 percent of the results. Architecture, design, and development are accomplished in a manner that makes it possible to add or customize services without changing the core code.

This allows a business to transform into an e-business with the ability to connect to a global business alliance or distributed knowledge and service marketplace.

This marketplace instantly provides information on products, data, and services, along with their values (some services might be more valuable than others), allows multiple parties/systems to negotiate multiple forms of collaboration, and contains sufficiently flexible levels of access security.

Businesses will be able to leverage their existing application assets by making them available to others via Web service and distributed technologies. Business frameworks, presentation factories, Web service, and distributed technologies (and much more) are discussed further in separate chapters.

SOFTWARE ARCHITECTURE AND INTEGRATION-READY SYSTEMS

The past decade has moved software architecture into the center of system development. It has also provided many (perhaps too many) examples of inefficient “quick and dirty” solutions prompted by a lack of integration among the three worlds of what, why, and how.

The latest buzzwords, industry trends, and components have often been used without necessity and without understanding that they are beneficial in specific circumstances under specific requirements. At the same time, some projects have not benefited from new standards and technologies, simply because developers have not been aware of their existence or have not had enough skills to apply them.

The importance of developing integration-ready systems and components is another lesson we continue to learn from the past.

The book teaches the reader to understand design patterns and architectural styles, to be able to see multiple sides of a system through the prism of industry standards and specifications, to communicate this vision via multiple architectural views, and finally to transform this vision into design and code that can make the “write once” dream a reality.

ENTERPRISE SERVICE COMPONENTS AND CHALLENGES

Wired and Wireless Telecommunications

Enterprise communications include telecommunication services as well as a variety of IT and data services.

Telecommunication services are competing for a limited market. (Though we called this market “unlimited” in the past, this clearly did not help matters much.) Telecommunication vendors offer new products and services that balance reliability, superiority of features, and price. Here is the client’s usual question: “Will your new program work with my old one?” In this case, “I am not sure” is not the right answer.

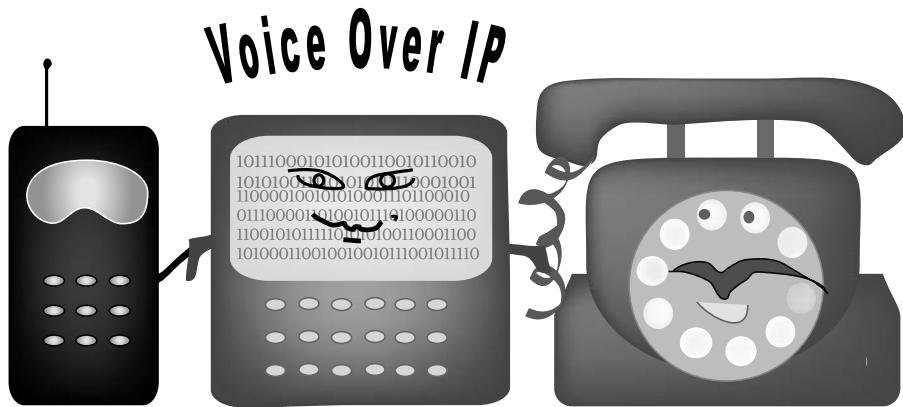


FIGURE I.5. Phone Generations Hand in Hand.

There are “9-to-5” employees who work only in the office and use fixed telephony. There is also a growing community of telecommuters: mobile employees who, for example, work one day in the office and four on the road, and who use wireless telephony. Corporations have multibillion dollar investments in Legacy PBXs that run regular telephony. This is the best guarantee of their longevity. Voice over IP and wireless communications will have to find ways to coexist in the mixed-enterprise telecom environment. Interoperability and a smooth transition can be achieved with a unified approach to wired and wireless application architecture.

The increasing quantity and variety of wireless devices sets new challenges for companies competing for multiple markets that run multiple wireless technologies. The winners will be those wireless Internet providers (WISPs) and wireless application service providers (WASPs) who can offer reliable connections and content services and optimize development solutions with a unified approach across multiple client devices.

Wireless clients are truly mobile. Client environments, locations, connections, and other parameters can be changed more frequently and drastically than those of fixed-wired network clients. We have to develop smart location-based services for our PDAs, cars, and robots equipped with GPS (global positioning system). We will have to switch from centralized, fixed computing to *distributed self-healing networks* with *flexible service redistribution*. We'll deploy *knowledge base-powered* application solutions: quick and smart adjustments to client locations, connections, and overall environment changes (Fig. I.5).

Consolidation of Multiple Data Sources

Multiple data sources were historically created in multiple departments for different and often unrelated purposes. Company mergers and the process of building Web umbrellas for company services bring a new focus to an old task. Data consolidation is becoming one of the highest enterprise business priorities. Enterprise Web applications require data integration and consistency. Data integration is especially relevant to telecom applications, which by their nature must be connected to multiple enterprise data storages.

Data consolidation is an expensive, nontrivial process in which XML and Java technologies can play an essential role. In Chapter 6 we consider a unified approach to multiple data sources (Fig. I.6).

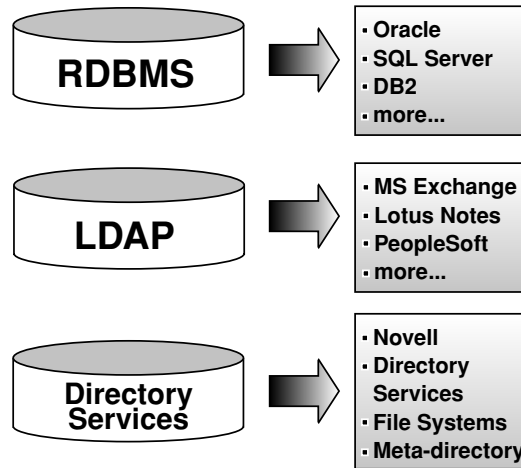


FIGURE I.6. Multiple Faces of Data Storage.

Natural User Interface and Knowledge Technologies

There are dependencies between the natural user interface and knowledge technologies. For example, current speech technologies are not quite ready to understand natural language speech initiated by a person. There are similar difficulties with machine translation from foreign languages. Can tasks like machine vision, speech recognition, and foreign language translation, which seem very different, have a common solution?

The first common denominator is easy to find: all these tasks belong to the field of data processing. The next move is to find common steps for all three activities. What do people do when they are engaged in image or speech recognition, when they translate text from Hebrew to English?

Can we build a computer program to perform similar steps? We know that data we usually keep in relational databases can hardly help us. What types of data are needed for the process?

Distributed yet connected, specific yet based on generic roots, different in themes yet non-conflicting: these are the qualities required for information to be inherently useful. Creating this quality data is extremely difficult; however, it has been proven possible.

We discuss architecture, design, and code samples that integrate knowledge technologies into enterprise applications and give the keys to a new breed of software—"softology," a mix of software and ontological engineering.

Because the knowledge engine is a new element that plays an important role in such architecture, I label it *knowledge-driven architecture*. It is very likely that this type of application will become the next big thing (after service-oriented architecture and event-driven architecture) in the near future.

This promising direction can raise the IQ level of our products; add more flexibility, intelligence, and common sense to applications; and even change the development paradigm by providing business experts with better tools that can eliminate several layers of the current development process.

Location-Based Services (Topology/Maps)

Topology data or geographical information systems (GIS) are necessary for mobile environments, facility management, military operations, and more.

Check your group members On-Line US Mountain time is 27-JUL-2002 15:07

No events are scheduled for today in your calendar.

Ben's News and Services:



E-mail

11 new E-mail messages, total: 78



Web Space for Personal Files

13.7MB used



Chat-n-Draw and other Services



Check your Calendar or Schedule a New Event



Check your (19) collaboration centers

About Us



To access your personal and collaborative accounts with wireless devices - point such a device to <http://IPServe.com/wml> (This requires a device that supports a WML Browser). Use OPTIONS to create your digital login/pin alias for convenient wireless access.



FIGURE I.7. Integrated Wired and Wireless Portals.

In a very peaceful example, GIS can be used to locate things such as office equipment and to optimize sharing of resources such as books, computers, facsimile machines, copying machines, or microwaves, and services based on their locations.

Topology data also can be used by robot-based moving services to determine hallway widths and produce the best or perhaps only way to move an item.

IT Provisioning, Corporate Portals, and the Distributed Knowledge Marketplace

IT provisioning and collaboration technologies, which are often implemented with corporate portals, increase productivity and encourage the creation of a distributed knowledge and service marketplace. There, data as well as services can be measured and can represent a new currency.

People produce knowledge and services every day. Unfortunately, most of the knowledge gained by individuals is rarely shared. We have no direct access to this greatest treasure of humankind. Distributed knowledge technologies can improve the situation, drastically decreasing the time between a service's supply and the market's awareness of it, providing a more precise estimate of data and service usability values, and introducing a new marketplace where information and service contributions are rewarded according to their values.

Collaboration or openness must be supported by security mechanisms. Multiple security domains with multiple member roles and multilevel privilege-based access are a necessity to encourage data sharing and active teamwork. Collaboration is not just about corporate employees working at corporate workstations. Wireless devices using WAP or speech technology (phone) clients, as well as regular workstations, will also be looking to access the rich content of enterprise data and services. The full content can be made available via corporate workstations while team members with thin client devices use a subset of the data and services (Fig. I.7).

INTEGRATION OF SOFTWARE AND KNOWLEDGE ENGINEERING

There is a gap in the current development process between initial business input provided by business people and the final implementation. The current process requires multiple transformations to simplify the complex world of reality into low-level program functions and tables expressed with current programming languages. This gap is filled with multiple filters/layers and sometimes multiple teams.

The shift to service-oriented architecture together with recent advances in knowledge technologies has paved the way for a new development paradigm.

In the new development paradigm, knowledge technologies play an essential role in the development process. Business domain experts will be able to directly participate in the development agenda by writing business rules and application scenarios in a more “natural” language. This will also help to focus developers (currently distracted by technological details) on the business aspects of applications.

Some time ago, we thought of the C language as a language for application development, whereas Assembly was the language for system development. It is about time to move the current programming languages, such as C# and Java, to the system level and make applications that are driven by knowledge-based rules and scenarios.

Knowledge-driven architecture is not a dream but a reality.

START WITH GOOD DESIGN AND REPEATABLE PROCESS

There is always a temptation to jump ahead several steps. Sometimes this is possible, but it is never safe. (This is my experience, at least, as a former mountain climber and an inventor.) The plan for this book is to learn the best current development practices in the software industry, climb close to the top, and then see the horizon. We focus on Object-Oriented Programming (OOP) and we also look beyond. There are Aspect-Oriented Programming and Generative Programming directions. OOP is not the end of the story.

Current technologies have brought significant power to developers. These technologies can be successfully used to implement good design ideas. They can also be used to unsuccessfully implement bad ones. How do you start walking in the right direction before you can see the end of the road?

Enterprise applications are often complex enough to offer us this challenge. Building a robust development process and building a team capable of following this process are probably the most important tasks of development. Good teams produce good products. It is appropriate, then, to start Chapter 1 with a discussion of teamwork and development processes and then to move on to technology.

Notes for Educators: AMA Teaching Methods

The AMA teaching methods are Activate, Motivate, and Assess.

ACTIVATE

Interact with the group and solicit each student's contribution. Engage your students in *collaborative learning* (let them work in teams) and *collaborative teaching*—give the stage to your students so they can review the material and share their best practices. One learns twice while teaching! Reward the best contributors.

MOTIVATE

Use all possible means to connect the course materials to the students' areas of expertise, thus increasing their motivation level. Motivation is the key to opening students' eyes and ears and unlocking their memories.

ASSESS

Facilitate a question-and-answer session, let students work in groups, and help them keep precise focus and timing.

Use the online assessment and evaluation forms periodically from the very beginning of the course.

This book provides a special section at the end of each chapter to reinforce the AMA methods.

The **Integrating Questions** are designed to help students build a thorough understanding of the relevance, relationships, and application of the content in the real world. To ensure

that students can relate course theories to the workplace, illustrate points with examples drawn from your professional experience and the experiences of the students.

The **Case Study** serves as a guide for student assignments. The instructor can modify and personalize assignments based on instructor–student interactions and early assessments of a student’s current and desired skills.

RECAP PRESENTATIONS AND MIDTERM ASSESSMENTS

Conduct two- or three-student recap presentations (ten to fifteen minutes each) starting with the second class. The content of each presentation is the student’s work on the Case Study–based assignments. The presentations may serve as midterm assessments. They also increase students’ motivation level and their understanding of the material (you learn twice while teaching).

DELIVERY FORMAT FOR ASSIGNMENTS AND PRESENTATIONS

It is recommended to format assignments and presentations as illustrated white papers using, for example, Web pages, MS Word, PDF, or PowerPoint with notes. The content can be presented in three levels: definition (high level), functionality and applicability (middle level), and examples of implementations (low level). These three levels roughly map the “why,” “what,” and “how” of the selected subject. Students will enhance their ability to clearly express their ideas and will learn the art of publishing and presentation with text, images, diagrams, and code extracts.

FINAL TEAM PROJECTS

Students may integrate selected parts of their weekly work assignments into team projects. This approach helps the student increase the quality of his or her work and consistently focus on the course materials.

Integration is not a copy-and-paste process. The team project allows students to exercise the downloadable software tool that comes with this book and helps them share their work with a distributed knowledge repository. The high visibility of a student’s achievement adds to the perfection of her or his work and makes it possible to directly tie students’ efforts to industry needs.

CHAPTER 1

Collaborative Engineering

Collaborative engineering is an important subject that is currently missing in schools, although some courses include a few of its aspects. I doubt I can cover it all at once, but I will try my best to *integrate* development technology and the development process under one roof, where integration-ready systems can be created.

Technology tends to fragment: to focus on pieces and omit the glue. The development philosophy that I associate with the terms of collaborative engineering helps keep a better balance between a narrow focus on particular components and their multiple connections to the rest of the world. The practice of collaborative engineering also assists in establishing a repeatable—yet flexible and constantly improving—development process. This is the foundation for integration-ready systems development.

What is collaborative engineering all about? It is about the development process in its organization, management, and methodology, integrated with innovative development technologies. I have had the privilege of teaching corporate developers and architects in the United States and overseas. Many times, I have had the amazing feeling that some of my brightest students, experienced developers, knew all the pieces of the puzzle and still could not start putting them together.

For example, almost every programmer knows one of the main design rules: Separate business from presentation logic. However, there are still more cases that break this rule than cases that follow it. One of the hardest questions is how to apply the rule properly.

Some developers believe that although theory is good, it is too costly for real life. This myth prevents many of them from developing the skills (in everyday practice) to quickly apply an existing design pattern or recognize a new one that will solve the problem “once and forever” and allow them to move on to other problems.

Guided by this myth, we win time and money on separate projects but miss the bigger picture, in which the industry pays a high price for overall inefficiency as well as data and service replication.

I think that by teaching separate pieces of the art and science of development, we omit the glue, the philosophy of programming that helps us find the right balance between “do it simply” and “do it once,” which sometimes leads to beautiful solutions in which the two rules are not in conflict but are happily married.

Software is built from big and small blocks selected by architects and developers. In the top-to-bottom development pattern, the big blocks are selected first; the developers then try to fill the holes with the smaller blocks. During this process, they often find that the original direction of the design was wrong and there is a need for remodeling.

How can you start walking in the right direction before you can see the end of the road? Even some good developers cannot begin a project until they can see all the details involved. Unfortunately, the scope of enterprise applications and time limits rarely offer this luxury to developers.

The right answer is a development process that has a framework yet is flexible – a process that is understood, followed, and enhanced by developers.

At this point, some readers may be thinking that all this talk about a process is management’s piece of the pie and not something developers need to worry about. Let’s read further and reserve our judgment. We will see later how much the management style and the development process contribute to overall results.

MANAGEMENT STYLE AS AN IMPORTANT PART OF THE DEVELOPMENT PROCESS: TRUE LEADERS VERSUS “PURE” MANAGERS

I do not even have time to read my own email reminders; of course I missed yours.

—From a management conversation (the terrible truth)

While working for multiple corporations as a developer or one of the key managers, I observed different management styles. In very generic terms, I would distinguish two major tendencies: “pure” management and true leadership. What is the difference between pure managers and real leaders? Some companies hire managers with little or no background in the field they are supposed to supervise. Their role is to provide and track plans and serve as a layer to smooth the edges among groups related to the plan.

However, project management is not just a Microsoft tool. It takes a leader to build a team, create a teamwork process, and promote best practices and collaborative technologies. It takes a leader to make decisions with creativity, not just through multiple-choice answers. Removing unnecessary layers and dealing with leaders instead of managers can greatly benefit projects, teams, and companies. I have learned this the hard way, working on multimillion dollar projects, consulting start-up companies, and teaching (and learning from) enterprise architects.

Shifting the weight of management toward developers enables those developers to grow and become leaders themselves. This approach focuses more on people than on current projects or tasks. It assumes (and this is the right assumption!) that the *main assets of a company are people*—and teams. The best teams produce the best products and services.

Workers (e.g., testers, developers, and system administrators) need to communicate and to translate to managers the essence of the work they are doing.

This takes minutes if a manager is a leader who knows the work. It takes hours for a pure manager who tries to learn a subject on the fly. The next time a worker wants to play a similar song with the manager, he or she has to start from the first music class—this note is called A, this is a C *sharp*—again and again. Even then, only a fraction of the information is retained.

Why? The orchestra conductor (the manager) is not a musician at all. (I know that many are, in a more literal sense.) There is no background where information can be stored. A pure manager often feels forced to choose between several opposing voices or lines in a document, and each voice or line sounds and looks the same as the others.

The management process often turns into a game for such a person, a game with the following rules:

- Use buzzwords, preferably generic (e.g., “stay focused!”) or at least technological (e.g., “scalable solution!”).
- Do not translate words into actions. Actions can be punished, because any action can be less than 100 percent safe, or incomplete, or not quick enough. Words are always right (“stay focused!”).

A leader owns the solution that he or she created. *This ownership feeling is at the heart of the development of any creative process.* (I just read that again, and it looks like a line from Ayn Rand.) In this context, *ownership* means responsibility to make and foresee changes, fix problems, and answer all questions related to the solution.

Pure management removes ownership feelings, destroys creativity, and makes people “come and go.” Every developer in the team is a leader who owns and has full responsibility for a specific area. Experienced developers with teamwork skills become the best managers or leaders.

On the opposite extreme of this spectrum is a leader who owns all the solutions, thus has a hard time delegating responsibilities. Such leadership can improve product quality in the short term (if this leader is the most experienced developer).

At the same time, this “babysitting” removes the ownership feelings of the others and prevents developers from growing on the job while making all the development tasks a sequential process funneling through a single leader-bottleneck. A leader should trust and delegate.

Product design and project management are interrelated. I recommend a practice in which team leaders are not only responsible for the part of the project delivered by their team, but also personally involved in that part’s integration into the overall project. Leaders keep teams and projects together as orchestra conductors do with musicians and music.

Where is the border between a leader and a pure manager? We all have elements of both, in different proportions. We act as pure managers as soon as we start talking about things beyond our expertise without investing time and effort in learning the field.

Being in a position to issue orders makes the pure manager rather dangerous. Investing time in active learning is hard—but rewarding—and develops a leader.

“I have no time to even read my email” is a phrase from a Dilbert book. It is still a very popular line in the management drama. Another extreme is “I have done much more complicated projects before.” Phrases like these are used to excuse a person from learning the specifics of a current task or project. Pure managers avoid specifics. This limits

their participation in a development process in which generic frames are filled with specific details.

DEVELOPMENT METHODOLOGIES: CAPABILITY MATURITY MODEL AND MORE

Some development organizations follow the Capability Maturity Model (CMM), some prefer ISO 9000 recommendations, and still others use Extreme Programming (XP) or Six Sigma methodology. None of these models suggests that it is an exhaustive recipe that covers all cases. The development process that fits your team and project is going to be built and enhanced by you. The process rules, or steps, should allow flexibility; they cannot be “final,” in Java terminology.

An Extremely Brief Overview of CMM

The CMM [1] can be used as the basis for diagnosing an organization's software processes, establishing priorities, and acting on them. One of the most important goals of CMM (and of ISO 9000) is the capacity to measure success and reliability of software processes. Measurement results allow for process improvements. Process improvements lead to a maturity level increase. CMM recognizes five levels of maturity in a software development process. Each level comprises a set of process goals that, when satisfied, stabilize an important component of the software process.

1. *Initial*. The process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics. Delivery dates for projects of similar size are unpredictable and vary widely.
2. *Repeatable*. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3. *Defined*. The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
4. *Managed*. Detailed measures of the software process and product quality are collected. Both the software process and the products are quantitatively understood and controlled.
5. *Optimizing*. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The International Organization for Standardization provides an ISO 9000 document that serves as a standard to certify processes and procedures. Whereas CMM focuses strictly on software development, ISO 9000 has a much broader scope: hardware, software, processed materials, and services. CMM provides many more details and measurement criteria.

The bottom line of both methods includes “document before acting,” “plan and follow,” or “say what you do, and do what you say.”

EXTREME PROGRAMMING: RULES OF THE GAME

Extreme Programming methodology gains more popularity every year. This section provides an overview of key XP concepts. (I can almost hear an impatient programmer's voice saying, "Another extremely brief overview? When are we going to see source code?")

Software development is not just about coding, as the art of painting is not just about the right paint selection. The heart of XP is a team of developers able and willing to communicate, plan, and design before coding. The team owns projects and processes and has fun planning and playing the project game by its own rules ("establish and follow"). A subset of XP rules is presented in Figs. 1.1 and 1.2. Does this look like XP?

SIX SIGMA

Six Sigma is another method that focuses on measurements of process capabilities and offers an integrated approach to shareholder value creation. Six Sigma suggests establishing seven to twelve measures in two areas: "critical to your business" (efficiency measures) and "critical to your client" (effectiveness measures). We select important measurement points that significantly contribute to overall business success.

Now we can consider an equation where business success depends on the measured functions. Some of them can serve as leading indicators that invite discussions such as "Why are these measures not moving in the right direction?" This enhances both discussions and critical thinking. Six Sigma helps to effectively break down complex processes into a matrix with multiple components and consistently works on component improvements.

Rational Unified Process

Rational Unified Process (RUP) is a method and a tool developed by Rational Company (currently a part of IBM, Inc.) that describes development as a four-phase process. In the *inception phase*, developers define the business outcome of the project. The *elaboration phase* considers basic technology and architecture. Developers deal with detailed design and source code in the *construction phase* and finally deploy the system in the *transition phase*. RUP offers software products that support developers as they walk through the phases.

Who implements the rules of the development process? People. This is the bottom line: You can select any methodology that fits your organization. Teams with their managers, leaders, and developers who understand and share the ideas of the methodology will make it work.

DISTRIBUTED COLLABORATIVE DEVELOPMENT

It is not an easy task to establish and follow a system, even for a single company. Can it work for a collaboration among several teams? Will this change the rules and the process?

Software communities working on common projects have recently suggested several answers to these questions. Two of the best examples are open source communities and the Java Community Process. However, there are still several factors that limit the success of collaborative engineering today.

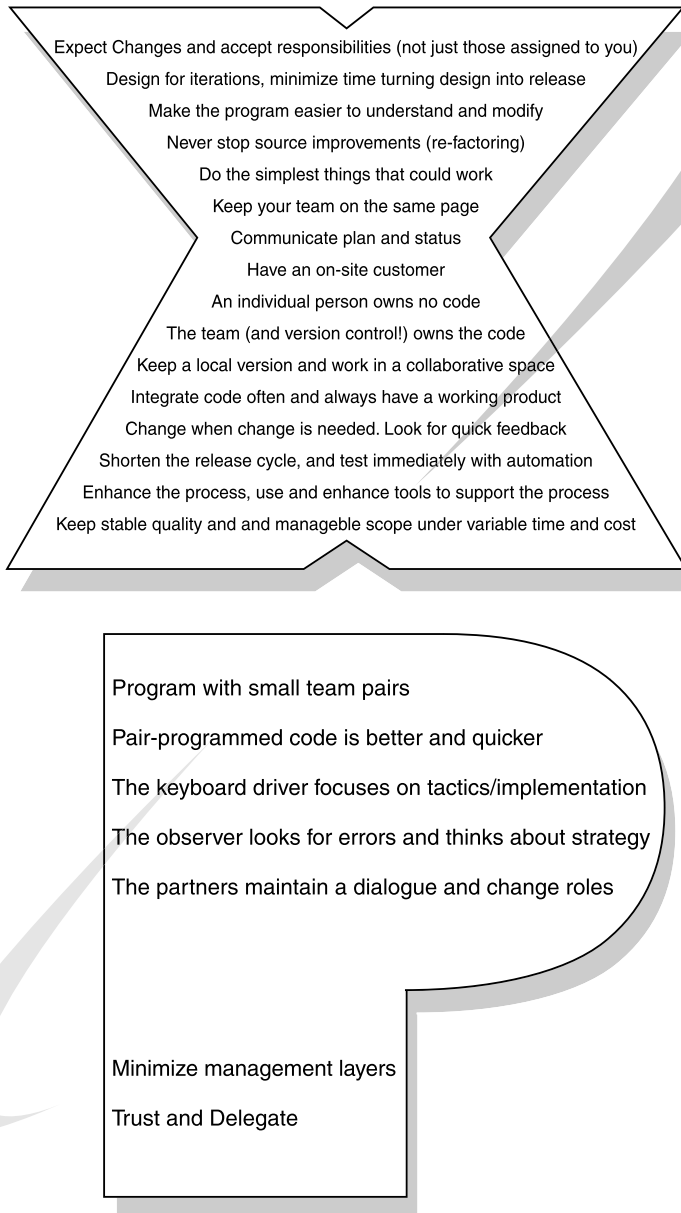


FIGURE 1.1. The Rules of the Game (XP).

The motivation for sharing is very low, and there is no flexibility in establishing the rules of the game: security, roles, access, and so on. The value of the data to be shared or exchanged is unknown, and a contributor rarely gets her or his reward because the contribution itself is not accountable.

For more than a decade, I have worked with remote students and development groups in trying to overcome these limitations. I came up with several innovations, described in a corresponding patent application [2] as distributed active knowledge and process technologies (DKT). DKT supports the process of distributed development and helps to improve processes while enhancing technologies.

Distributed Knowledge Technologies

Existing on-line collaborative services allow users to share only a limited set of data types, usually restricted to messages and files, with the rare addition of a shared organizer or other similar service. This narrows collaborative actions to a small number of fields and introduces limitations on the scope of possible collaboration and data sharing. Though some users are satisfied with restricting their collaborative efforts to sharing files and sending group messages, such systems are often insufficient in scope to allow for efficient workflow in a real collaborative setting.

Existing services on the Internet also limit their collaborative structure to data objects and exclude processes. As a result, the large amounts of data that can accumulate in a group knowledge base cannot be mapped to better processing methods. As the number of data objects increases, it becomes more and more difficult to use the information contained in them to efficiently accomplish goals.

Current system structures do not permit users to collaboratively add objects of unknown data types and a service to process a particular type of data to best suit the goals of a group. They also prevent the creation and implementation of preprogrammed processes, services, or scenarios for distributed processing. This further limits collaborative efficiency.

Existing systems own and fully control their collaborative environments. This limits collaboration to a single system and does not permit systems to share data or other system elements. Data, process, and service sharing between systems belonging to different organizations is an even more complicated issue because there currently is no way for a system to determine and specify elements appropriate for free public sharing, elements that are to be shared on a pay-per-use basis, and elements that are to be exchanged for others of equal value.

Last, current online collaboration is limited by the unwillingness of users to share their data. Even in a collaborative setting, users rarely desire to make their data available to all members of their group, and they make adequate security a condition for sharing information.

The backbone of any online collaborative effort is security, and the current methods of assigning access privileges as a way to make specified data objects available to the appropriate viewers are inadequate.

Existing systems allow limited role-based privileges for all collaborative data. A common system has a limited number of privilege levels (in most cases, two). In such a system, if a user's profile defines her as an "administrator," she has read, write, and delete access to all group data.

If a user is defined as a "member," he can read and add messages but cannot edit or delete existing messages. This kind of system is limiting and does not encourage data sharing because it does not give users control over their data. Users cannot create new custom roles on the fly, cannot select who has certain kinds of access to the information they choose to share, and must provide the same level of access to all members within a privilege class.

The willingness of users to share is also limited by their knowledge of other elements inside and outside the user's system and of their values. It is important to know how valuable a

particular service or data object is from a user's point of view. Current systems provide only the number of times a file has been downloaded and selected positive comments by current users. A new mechanism is needed to provide and update more meaningful usage and viewer response information inside the system and between systems.

DKT covers a need that exists for collaborative systems and permits increased flexibility in the types of data and services that can be shared. It allows data, processes, and services to be created and modified within the same collaborative framework and permits the mapping of appropriate data to these processes. DKT systems can notify interested parties on available objects, processes, and services and provide dynamic evaluation of data and services based on their usage.

DKT systems can negotiate several forms of collaboration and establish rules of access for internal members and outsiders (e.g., pay-per-use, and value-based exchange), using sufficiently flexible levels of data security to foster online collaboration. Sharing can be an enjoyable habit and an eye- and mind-opening experience (not a medical term).

This new mechanism to value data and services creates an environment of active participation in which each contribution is accountable. This accountability motivates contributors to enter this new marketplace, where knowledge and services play the role of virtual currency. There are many ways of transforming virtual currency to hard currency. What is the exact mechanism of this transition? Participants will find or invent a good one.

There are plenty of "how to" details and usage descriptions of DTK systems that might be too boring in the context of this book. They are very well explained in the DKT patent application published by the U.S. Patent and Trademark Office [2].

24×7 DISTRIBUTED DEVELOPMENT PRACTICES

Especially important for an international development team with members distributed over the globe and working twenty-four hours a day, seven days a week, are "24×7" distributed development practices.

Is it possible to achieve coordination, understanding, and cooperation among multiple teams working in different time zones, speaking different languages, and living in different cultures? Can distributed development be cost effective and highly efficient at the same time?

Our virtual team was created as a side effect of several years of on-line training. It was a very attractive concept: to extend time and resources on a global scale. Learning the ABCs of important team member qualities/abilities and developing a core of "black belts" (in Six Sigma terms) for each team was, though very beneficial, quite a challenging task.

Collaborative technologies make the development process highly visible for every team member as well as for the management team. This visibility helps establish and reinforce teamwork rules. The system forces developers to provide daily and weekly plans and status reports that improve analysis and design quality. I say "forces" because the plan and status jobs are the least loved by developers. DKT makes them unavoidable habits. A notification engine working with the task management systems provides an action-reminder if a needed action has not yet been taken.

Global collaboration encourages open expertise exchange under a partnership umbrella. A very valuable benefit is that we can achieve much more working together than we can achieve separately. Network specialists from Hong Kong; a "Fifth Dimension" scientific group from Scandinavia; SZMA (oil and gas factory automation international integrators) engineers;

computer science researchers from St. Petersburg University and Institute of Information Technology in Russia; JavaSchool students and consultants.: these are some examples of teams collaborated in the distributed knowledge environment.

This book includes design and code samples and offers a tool that allows you to build your own system powered by a knowledge engine, support your own development process, establish your own rules of sharing, and, if you wish, connect to other systems and join in collaborative engineering.

STEPS IN THE PROCESS

What Is Your Development Mode: Proactive or Reactive?

We are coming back from the heights of management to the specific steps of the development process. Here is a quick quiz related to your organization’s habits toward time distribution on a software project.

Which line (1, 2, or 3) represents the percentage of project time your organization allocates to analysis, design, code, and testing?

	Analysis	Design	Code	Testing
1.	40	30	20	10
2.	10	20	30	40
3.	25	25	25	25

There is no right or wrong answer to this quiz. Your selection sheds some light on your organization’s development habits. My experience shows that for bigger projects, shifting gears to line 1 helps to improve quality. However, for really small projects, number 3 works quite well. I have noticed that the second answer is often, though not always, associated with start-ups or organizations that are on the learning curve using “let-me-try-it” paths. (I hope I do not sound critical.)

Amazingly enough, all three ways can work very well for a team that uses one of them as part of a repeatable development process. Such a team usually understands where steps can start and stop and is able to recognize cases in which it makes sense to move along in the cycle. There is no single good answer. This is about your organization development style.

The object-oriented approach to application development is currently the leading approach in the software industry.

I will describe some basic steps of software application development with this approach; I have to add that the steps and their sequence can vary. I am sharing my experience, and I know that there is more than one good methodology, as there is more than one good solution to almost any problem. The most important thing is to have a system and consistently follow it: “Plan and follow” or “say what you do and do what you say.”

Basic Development Roles

Developers (or a single developer for a small organization or a small project) play different development roles at major phases of the development process. System definition is usually performed by **system analysts**, **subject matter experts**, and **architects**. Architects and

designers are responsible for system analysis and design. Developers participate in the implementation and deployment phases. **System developers** or service providers are different from **application developers**, who use provider services to create a higher-level system for end users. This separation is not always obvious. Service providers also have end users, but in most cases, their end users are application developers who can use their services.

It is possible that the deployment process requires **system integrators**. This role might disappear as more integration-ready products come out after this book is published. (I know you are smiling.)

Architects do not participate in the deployment process but are responsible for the deployment plan, which is usually expressed in the deployment diagram.

This is a very generic definition that can be customized to fit your development process specifics. For example, according to Sun Microsystems recommendations, Java 2 Enterprise Edition (J2EE)-based development involves the following roles:

- **Tool provider** (for example, Borland): provides development tools
- **Application component provider**: a generic name for Web component providers that develop Java server pages (JSPs), tag libraries, servlets, and related business classes located in the Web container; Enterprise Java Bean (EJB) developers; and Web page designers
- **Application assembler**: uses components developed by application component providers to assemble them into a J2EE application component
- **Deployer**: responsible for application deployment in a specific environment
- **J2EE product provider**: implements a J2EE product
- **System administrator**: provides the configuration and management of the system
- **Quality assurance (QA) engineer**: provides integrated testing.

Most of the development methods obligate developers to provide “unit testing” of the part delivered by the developer to a current system release. But system-level testing often requires a special environment that emulates production and special efforts.

The Role of the Software Architect

Software architecture is the highest level of a software system definition. A **software architect** is the one who actually provides this definition. The architect starts with user requirements and separates them into two categories: **functional requirements** that refer to system tasks and **nonfunctional requirements** that refer to system efficiency, scalability, flexibility, and other “-ilities.” The architect participates (to some degree) in the analysis and design processes but does not go into implementation details; he or she focuses on the larger picture of the system instead.

It is important for the architect to negotiate with the client about the metrics (not just words) and priorities of requirements criteria. For example, the “must be scalable” requirement should come with some rough range numbers (e.g., expected number of users).

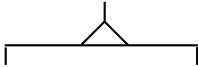
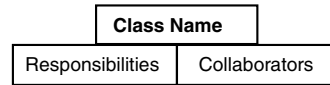
BASIC STEPS OF THE DEVELOPMENT PROCESS WITH AN OBJECT-ORIENTED APPROACH

The basic steps of the development process are shown in Fig. 1.3. The rest of the chapter carefully moves over these steps.



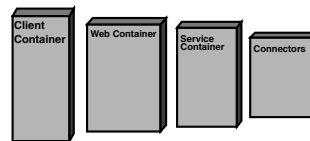
1. Capture User Requirements into a set of Use Cases

2. Play cards (CRC) to Recognize Classes



3. Form Object Model Diagram; define the relationships of the Classes

4. Architecture design: find playgrounds (tiers) for your objects



5. Recognize reusable services and shared data

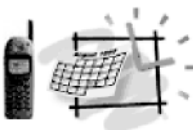
6. Define the screens and user interface with Human–Computer Interface experts. Strive for consistency and simplicity.

7. Define the date and scope of the first deliverable, targeting to prove the basic concepts and to get feedback from the client on the very early stages of the development



8. Use Style Guidance while coding and review the code regularly (this also keeps the proper level of communication for a team).

9. Share the first results with domain experts. Make corrections and outline the plan to finish and fine-tune the project.



Employ technology to establish a proper level of documentation and communications in the team. Move from archives to an active repository of the shared parts of the project.

FIGURE 1.3. Basic Steps of the Development Process.

Start from User Requirements and Transit to Use Cases

Start with interviews of your clients, who are (hopefully) the domain experts. Your goal is to understand business rules and business user requirements. This is a proven way to smoothly come to an agreement with the customer and the users on what the system should do. The next step is the transition from user requirements to use cases.

Split an application into a set of use cases, or scenarios describing subtasks of the application. Yvar Jacobson's methodology is used to capture user requirements in a set of use cases [3]. Each use case describes a scenario, a flow of events initiated by an "actor."

An actor represents anything that interacts with the system. Written descriptions of use cases are complemented by process diagrams. This step is the first attempt to transition to objects from plain human language that describes "what to do" in terms of events and conditions. The user requirements record is a "no-objects" document that may be ten or ten hundred pages long. It provides a high-level definition of desired system functionality and can provide a list of conditional actions the system produces. The use cases diagram breaks this flow of words and events into subtasks or scenarios. Main objects or actors appear from the background and start playing their roles (at least on paper).

The following example of user requirements reflects one of the difficult situations that must be resolved by developers (almost a true story).

Black Holes, Inc., was a start-up company that targeted travelers by providing maps and directions to the least known and visited places, or "black holes." Unfortunately, the company did not meet annual revenue expectations. (The initial orders that looked so promising for investors were submitted by the company VP of marketing during his research work in the competing territory of the Bahamas.)

The board of directors took some measures to change the situation. It was decided to cut 75 percent of the development team staff (there used to be four of them) and introduce incentives to correlate each employee's paycheck to his or her actual work. The hourly workers' paychecks would depend on the number of hours worked. All the managers would continue to receive their salaries, but would get no bonuses while the company operated in the red. It was confirmed once again that the holes were extremely hard to sell. Therefore, the salespeople would be paid an hourly wage plus commission to stimulate hard work. Three new positions were opened to support the project: an IT director to improve the communication of management decisions over the network; a VP of travel arrangements to focus the development team on two new tasks (searching for "black holes" and optimizing travel routes); and an order administrator to support Web client registration and privileged access.

Of course, the changes had a serious impact on the "chess match" between the management and development groups (Fig. 1.4). It was over. Ben, who represented what was left of the development team, did not play chess. He took the silver (a trip to one of the "black holes") while the management team was awarded the gold. *The management team's spirits were extremely high, and the project started with the motto: "Each employee should be accountable. Failure is not an option."*

We can find the main actors and associated tasks/scenarios on the use cases diagram in Fig. 1.5. The Unified Modeling Language (UML) is a formal set of graphical notations that helps visually express object-oriented realities (models) of software applications. Note that actors are associated with objects and from a semantic point of view, are nouns, whereas scenarios introduce the desired functions with names that start with verbs.



FIGURE 1.4. “Team Players.”

Object-Oriented (and/or Aspect-Oriented) Analysis—Make a Transition to Objects and Classes

From this point, we may concentrate on Use Case scenarios and then proceed to objects. After describing what needs to be done in terms of events, we start the transition to the object world; we now try to concentrate on data and pay less attention to the flow of events.

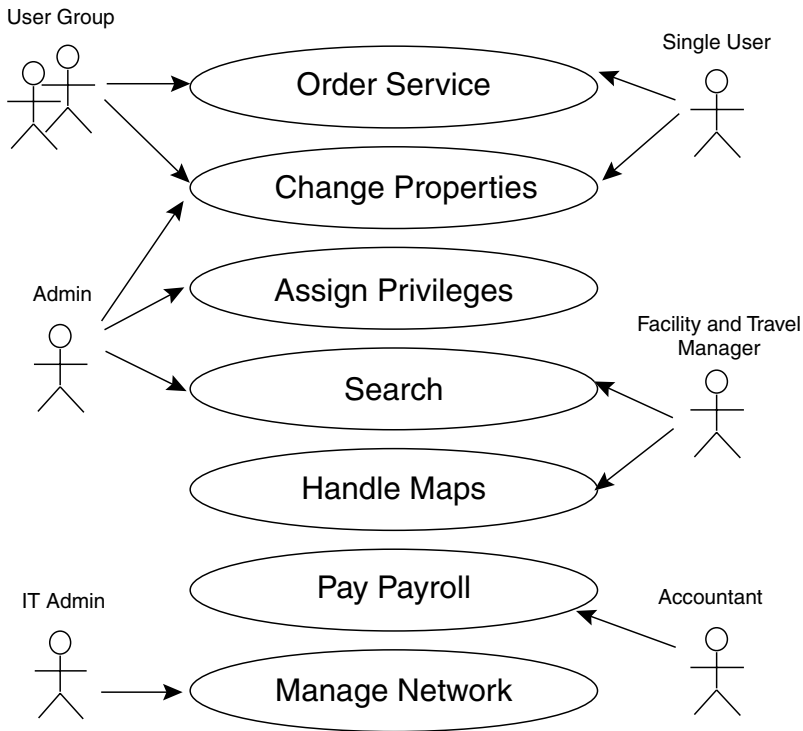


FIGURE 1.5. Use Cases.

It is definitely a different view of the user requirements; if you want, it is a different view of the world.

User requirements and scenarios usually describe a set of functions and a flow of actions produced. This approach represents a dynamic view of the application. It used to be the dominant approach supported by flow charts that showed structures of functions and conditions. This development practice was established in the 1970s as structural programming.

The programs started with the main routine that checked conditions and called the proper functions, which were easily visible on flow charts. Sources, such as the original flow charts, looked like this: “if THIS—call THAT; else, if THAT—call THIS” (Fig. 1.6). This style of programming was named “structural” because the main idea was to keep programs readable or “structured,” keep functions or subroutines in separate files, and avoid the temptation to put all the code in the main routine.

Where are those beautiful days of program simplicity? I am afraid that time will never return. Software became increasingly sophisticated in the 1980s while trying to model more and more complicated tasks of the real world. The number of details associated with a single task exceeded the capacity of a single brain, and a new abstraction was born: the world of object-oriented concepts. These concepts shift our attention to static parts of the application. We learn to concentrate on data objects and model system behavior as the collaboration of these objects.

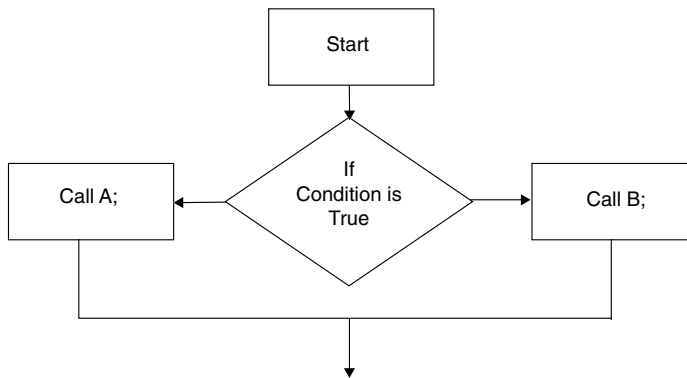


FIGURE 1.6. The Good Old Ways of Structural Programming.

Discussion about the static and dynamic view of the world started at more than 2,000 years ago:

The world is a process that never stops. —Heracleitus, 540–480 BC, (the dynamic view of the world)

The world is a composition of objects called atoms. —Democritus, 460–370 BC, the first object-oriented philosopher.

Here is your self-test: Who was right in this ancient debate?

1. Heracleitus
2. Democritus
3. Both of the above

(Hint: Both views, static and dynamic, are important and complementary.)

While transitioning to an object-oriented approach, we use the new abstraction “CLASS” to classify objects. An object is an instance of a class. How can we identify good objects—candidates for classes?

Look at the application through the client’s eyes, forget about programmers’ tricks, and concentrate on client objects. Look for repeatable objects, meaning those that are reusable in your application and complicated enough to deserve to be new data types. Look also for repeatable behaviors of these objects. If such objects can be grouped, a group or set of these objects can be considered a possible class.

The classes/responsibilities/collaborators, or CRC, approach can be used to identify classes. Any noun from a user interview or written business requirements can be considered a potential candidate for a class. Forget about implementation details for now.

A brief summary:

- Look at the application through the client’s eyes.
- Classify objects.
- Ask basic questions:
 - What objects are involved?
 - What are the responsibilities of the objects involved?
 - What are helpers/collaborators of the objects involved?

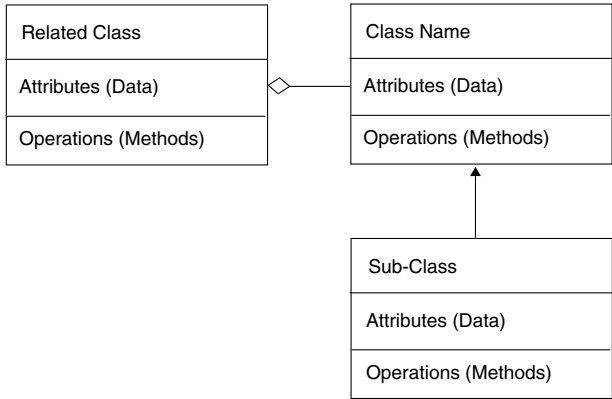


FIGURE 1.7. UML Notations to Describe Classes.

Classes allow us to encapsulate common object properties and behaviors. We can then hide implementation details beneath the surface of public methods or interfaces.

The main object-oriented concepts are encapsulation, inheritance, and polymorphism. The most important one is encapsulation. An object, according to this concept, is a self-contained set of data and functions or methods that operate on that data.

Each object represents a function, feature, or component of a larger system, much like a brick in a wall. The object contains all of the data concerning its function, along with a set of functions to access that data. The data is generally *private* to the object, but may be declared *public* if necessary. The “outside world” will access the data using the object’s functions or methods.

Classes define the data (attributes) and the operations (methods) of the object. An object is said to be of a particular class. A class is like a blueprint, or a plan, for the object. It defines the data and functions to be included in the object. A class is similar to a structure but contains functions (methods) along with data. In Fig. 1.7, we see an example of the UML notations that define a class.

Use UML to Capture and Communicate Analysis Results

We use UML to show classes with their relationships in an object model diagram (OMD). If we want to declare a class that represents a person, we might use examples from C++ (Fig. 1.8) and Java (Fig. 1.9) class definitions.

Another important concept of object-oriented programming (OOP) is inheritance. Inheritance is actually an extension of the encapsulation concept. We can find an example of inheritance in the payroll scenario for the Unbelievable Holes start-up company: “The hourly worker paycheck depends on the number of hours worked. All managers receive their salary regardless of sales. Salespeople are paid an hourly rate, plus commission to stimulate hard work.” We can clearly see three types of employees: hourly, salary (management), and sales.

When we look carefully into the set of scenarios in the use cases, we can see more actors, potential classes, related to the employee class. Both a single user and user groups can order services and change account properties. Single users and user groups have one thing in common: we call it an “Account” class.

We can also think about an employee as a potential user. Taking all this into account, we draw these classes, showing their inheritance relationships. The object model diagram


```
// Class CPPUser Example
class CPPUser {
    private: // data
        char name[30];
    public: // methods
        void setName(char* aName) {
            strncpy(name, aName;
        }
        char* getName( ) {
            return name;
        }
}; // ! semicolon at the end of class

// Example of usage within another class:

// Define a CPPUser object
CPPUser user = new CPPUser();
//set name = "Peter" to the object
user->setName("Peter");
```

FIGURE 1.8

extraction presented in Fig. 1.10 is just one example of modeling. The same scenario can be modeled differently.

User requirements, not just imagination, should lead to the proper model selection. For example, in our example, the payroll task requires specific properties, such as the number of hours worked and the hourly rate for hourly employees. If the user requirements included item shipment to users, we would need to add a shipping address to the user properties, and so forth. There is often more than one good model for the task. If you have a choice, select the simplest one. It is very possible that later on, more details will be available and the model will be changed.

If your object model's bad—fix it, don't blame my dad

—Ben Zhuk, Web Genius, Inc., president, yet another object-oriented philosopher

Talk in Java and C++ about Employee in the Payroll Scenario

Figure 1.11 illustrates inheritance in the payroll scenario. Figure 1.12 actually delivers the same message in formal UML language.

Once a class is defined, it may be used as a model for other classes. This idea can be expressed using the C++ syntax *class Child : public Parent* or the Java syntax *class Child extends Parent*. The new *Child* class inherits all of the attributes and methods of the *Parent* class. Additional attributes and methods may then be added to the *Child* class without affecting the *Parent* class. The *Child* class is derived from the *Parent*.

```

//JavaUser Class Example
/**
 * The JavaUser class is an example of a Java class with data and
 * two methods
 */
public class JavaUser {
    // data
    private String name;
    // methods
/**
 * The setName() method assigns name
 * @param aName
 */
    public void setName(String aName) {
        name = aName;
    }
/**
 * getName() method returns the name
 * @return name
 */
    public String getName() {
        return name;
    }
} // There is no ";" at the end of class

// Example of usage within another class:

// Define a JavaUser object
JavaUser user = new JavaUser();
// set name = "Peter" to the object
user.setName("Peter");

```

FIGURE 1.9

Here is another example of inheritance that introduces the classes *Grandpa*, *Mom*, *Dad*, and *Me*, presented in Fig. 1.13 with code and pictures. Notice that the class *Dad* is derived from *Grandpa*. We will declare one more class now in Fig. 1.14.

The class *Me* will end up with Mom's mouth, Dad's eyes, and Grandpa's big ears, because the *Dad* class was derived from the *Grandpa* class. In this example, freckles are the attribute that exists only in the class *Me*. This example only deals with *data* inheritance; the same situation would occur with any methods (functions) that were contained in the base classes.

Note that the freckles (as well as the eyes in Java implementation) in the class *Me* are defined as *private*. These attributes will not be inherited by possible subclasses (or children) of the class *Me*. Protected or public access allows inheritance. At the same time, such access types open up objects and, in a way, break encapsulation. This is especially true for public access that exposes data to the outside world.

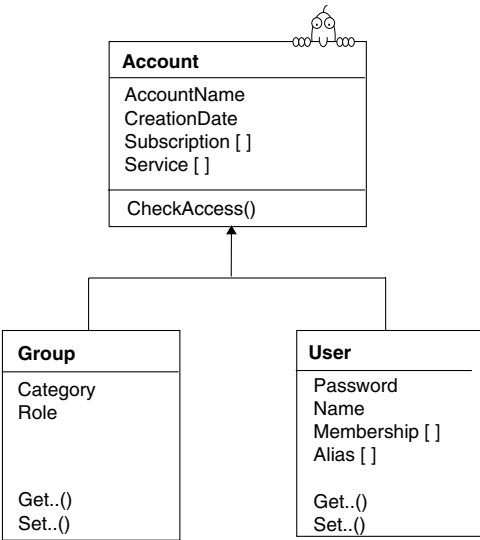


FIGURE 1.10. User and Group Accounts—Objects Model Diagram.

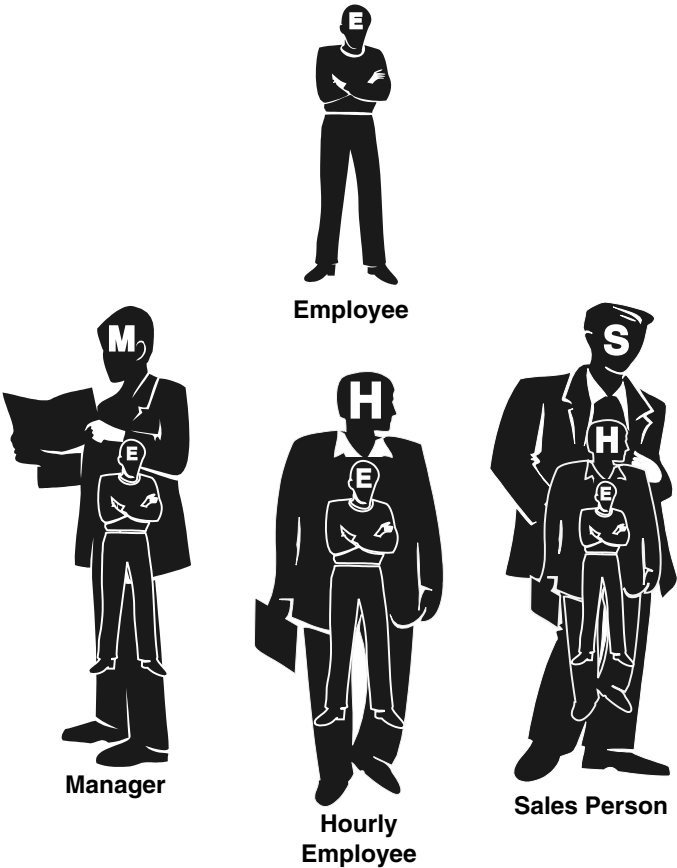


FIGURE 1.11. Inheritance: *Employee* and Potential Subclasses.

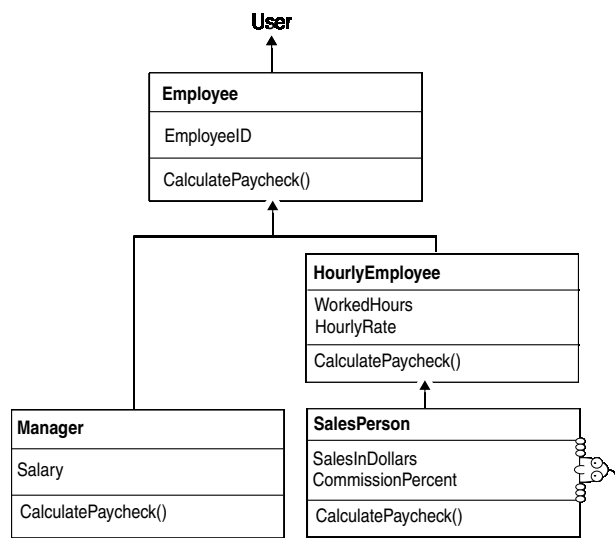


FIGURE 1.12. User, Employee, and Subclasses—Object Model Diagram.

The basic rule of OOP is to keep data private while providing public methods to access the data. Inherited methods can be redefined (or reimplemented) in subclasses that create a platform for another OOP concept: polymorphism.

Polymorphism means the ability to take multiple forms. In OOP, this also refers to the ability of an object to refer at run-time to a type (or a class) of the object to select proper methods. You can design for polymorphism and benefit from its power.

If we wrote a procedural code for the payroll scenario, we would check on the type of employee and use different methods. Procedural code shows up with its clumsy “if-else” statements everywhere. For example, we would write here:



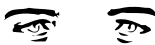
<pre>//C++ Implementation public class Grandpa { protected: BigEars ears; };</pre>	<pre>public class Mom { protected: Mouth mouth; };</pre>	<pre>class Dad: public Grandpa { protected: Eyes eyes; };</pre>
		
<pre>//Java Implementation public class Grandpa { protected BigEars ears; };</pre>	<pre>public class Mom { protected Mouth mouth; };</pre>	<pre>public class Dad extends Grandpa { protected Eyes eyes; };</pre>

FIGURE 1.13. Inheritance.


<pre>//C++ allows //multiple inheritance class Me: public Mom, public Dad { private: Freckles freckles; };</pre>		<pre>//Java doesn't support //multiple inheritance //notice that class Me //inherits from Dad ONLY public class Me extends Dad { private Freckles freckles; private Eyes eyes; //not from Mom };</pre>
---	---	---

FIGURE 1.14. It Runs in the Family.

```
If(employeeType == 1) { // this is pure procedural code
    calculateManagerSalary();
} else if(employeeType == 2) {
    calculateHourlyCheck();
} else // etc., etc., etc.

In OOP we can write the same thing cleanly and easily:
// Get any Employee: Manager, Hourly, or Sales
Employee employee = getEmployee();
// The system will use a proper method of Manager, or Hourly, or Sales
employee.calculatePaycheck();
```

The proper *calculatePaycheck()* method will be picked by the system at run-time, depending on the object class type. This powerful feature does not come cheap. You need to do analysis upfront and design your classes with polymorphism in mind.

What does it mean to design for polymorphism? Create a base (parent) class, or an interface, and a set of derived classes, and teach them to behave. This means implementing polymorphic methods: methods with the same names and arguments that have different implementations for different classes. Then, just call one of the objects or instances of derived classes and say “object.doTheJob().”

You waste no time figuring out which object type is being called; no *if-else* is needed. The object knows about its own type and behaves accordingly. For example, given a base class *Employee*, we defined different *calculatePaycheck()* methods for derived classes, such as *Hourly*, *Manager*, and *Sales*. Now we can take any employee object from the list and provide the line: *employee.calculatePaycheck()*. No matter what type of employee is selected, a proper *calculatePaycheck()* method will apply to return the correct results. Any true OOP language offers polymorphism. (Even Visual Basic in its latest version offers polymorphism.)

LEARN BY EXAMPLE: COMPARE OOP AND PROCEDURAL PROGRAMMING

OOP is not easier than procedural code, especially at the beginning of your learning curve. It pays off later on.

Here is another example illustrated with C++ code. Imagine that you write an application related to factory workers and inventory items. When you write a procedural program, you make a flat set of procedures:

```
--setWorkerName(*name);
displayInventoryItemPrice();
```

It is relatively easy to oversee code for two objects. But when the number of objects grows, the increasing number of procedures makes the code almost unmanageable. In OOP, you create objects and encapsulate (hide inside the objects) their properties, and the methods that can manage these properties. Your procedures are now distributed in several classes that provide their behavior. Your main method that drives the application will immediately shrink from your big, flat procedural surface down to a set of objects that can be introduced and then start to *behave*. In the following example, you create the *Worker* class and the method *setName* and then the *InventoryItem* class and the method *displayPrice()*.

Now you are dealing with a much smaller number of objects than procedures. You can cut your main method from a fat *if-else* scenario into a couple of lines that introduce your objects. Your objects have methods and properties. They behave.

```
Worker *s=new Worker(); // create a worker object
s->setName("Jeff"); // set the name
InventoryItem *i=new InventoryItem();
    // create an InventoryItem object
i->setPrice("10");
i->displayPrice();
```

Yes, we produce more code overall, but this code is more understandable and manageable than procedural code, especially when the number of objects increases. Notice that we hide, or encapsulate, data (like price) and methods (like *setPrice* and *displayPrice*) into the *InventoryItem* class.

Below is another example. This time, we consider an inventory program where you can perform one of a set of operations, such as add, delete, or change, on inventory items. For simplicity, we offer these choices from a command line menu. Your main (testing) procedure can be outside of your other classes. The main class (or procedure) has a menu.

```
// main function shows the main program menu
int main()
{
    InventoryMenu *menu = new InventoryMenu();
}
```

In the *InventoryMenu::InventoryMenu()* constructor, you display the menu and call methods from your menu that create (or take from a file, or take from a linked list in memory) your *InventoryItem* object. For example:

```

/**
 * constructor-menu offers several choices and calls the proper method
 * to operate on items
 */
InventoryMenu::InventoryMenu() { // constructor-menu
    cout << "Please select one of the operations below" << endl;
    cout << "Press 1 to add a new item to the inventory" << endl;
    // more choices
    cin >> choice;
    if(choice == 1) {
        addItem();
    } else {
        // more lines
    }
} // end of constructor
/**
 * addItem() method creates a new item and calls its add() method
 */
InventoryMenu::addItem() {
    InventoryItem *item=new InventoryItem();
    item->add();
}

```

The *InventoryItem* object has the necessary properties (*itemName*, *price*) and does the necessary operations on these properties. The properties and methods are defined in the *InventoryItem* class. For example:

```

InventoryItem::add() {
    // present main menu
    cout << "Please enter item name:" << endl;
    cin >> itemName;
    cout << "Please enter item price:" << endl;
    cin >> price;
    cout << "Thank you" << endl;
}

```

OOP Has Its Limits

Does this mean that OOP solves all programming tasks and can treat all programming diseases? There is a temptation to say yes. But this would not be a true statement.

OOP brings an abstraction that can be overkill for some simple tasks. It works well on sizable projects with multiple objects spread over a huge surface of requirements. We group objects, finding common properties and behavior or concerns, and provide common implementations called *classes*. These implementations are based on a one-dimensional view of the common concerns most often related to a business domain.

We can broadly classify task requirements into core *module-level requirements* and *system-level requirements*. What if system-level requirements crosscut many core modules? For example, a typical enterprise application comprises crosscutting concerns such as authentication, logging, resource pooling, administration, performance, and storage management.

This multidimensional view of the application presents a real challenge to architects and designers. Current OOP-based implementation techniques tend to implement these requirements using one-dimensional methodologies, forcing implementation mapping for the requirements along a single dimension. That single dimension is often the core module-level implementation. The remaining requirements are tagged along this dominant dimension. In other words, the requirement space is an *n*-dimensional space, whereas the implementation space is one-dimensional.

Can you see the problem?

Several solutions for this problem already exist: *mix-in classes*, *design patterns*, *domain-specific solutions*, and *aspect-oriented programming* (AOP) [4]. AOP strives to cleanly implement individual concerns in a loosely coupled fashion and combines these implementations to form the final system. The module unit in AOP is called an *aspect*, similar to a class concept in OOP. We do not go to the horizons of AOP (at least not now), but it is good to know that they exist.

OOP is a proven way to create multiple abstractions—layers that filter complexity of application requirements down to simple functions and data tables. Unfortunately, almost every layer requires its own art of creation, and creating an application becomes a multistep project with a growing distance between initial ideas and actual implementations.

As you can see, OOP is not the end of the story. At the same time, it is a very essential part of almost any enterprise development today. (See Chapter 5 for a new development paradigm.)

UML NOTATIONS

Many programming languages support OOP concepts. There is also a graphical support provided by the UML. The UML helps communicate design ideas to developers. There are also some tools, such as Rational Rose, that are capable of generating code out of UML diagrams and notations. Figure 1.15 shows some examples of UML notations describing classes and their relationships.

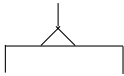
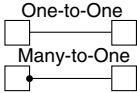
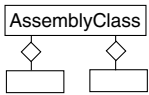
To describe Classes in UML, we use three sections: Class Name, Attributes (data structure) and Operations (Methods)	Class Name: Attributes: Operations:
This notation is used to reflect the Class inheritance hierarchy, or the relationships between the base and derived classes	
We use this notation when we want to outline associations between classes: One-to-one, or Many-to-one.	
This notation is used to display class aggregation	

FIGURE 1.15. UML Notations to Express Relationships.

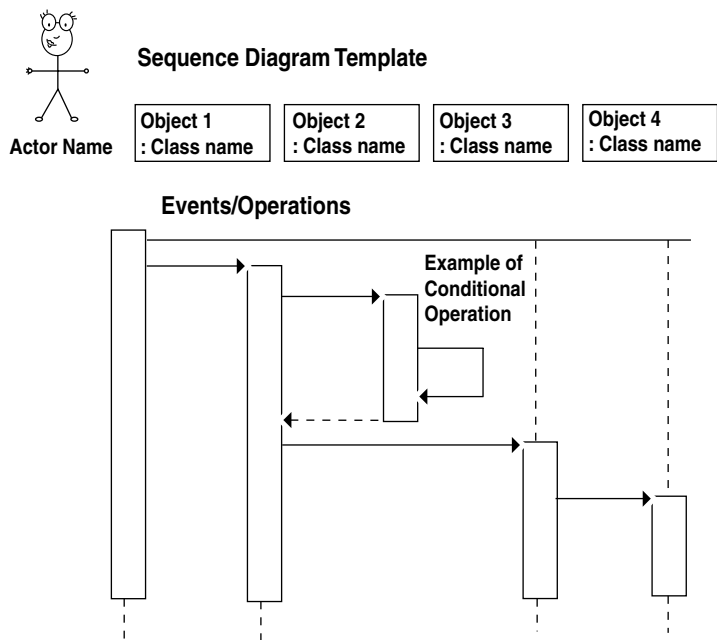


FIGURE 1.16. UML Notations to Express the Sequence of Events.

We already looked into an example of the OMD presented earlier. The OMD covers the static side of the design while the sequence diagram shows its dynamic side. Figure 1.16 presents the basic blocks we can find in almost any sequence diagram. The sequence diagram communicates the sequence of events and operations and participating objects. Figure 1.17

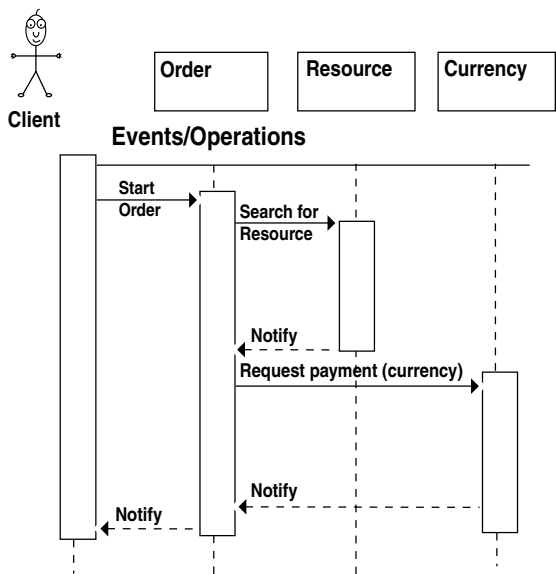


FIGURE 1.17. Service Order Sequence Diagram.

presents an example of a sequence diagram for the service order use case when the subject of the order is a physical item that can be located in stock.

The *OrderWindow* object sends an *initialize* message to the *Order* object, which in turn sends an *initialize* message to each *OrderLine* object.

The *OrderLine* object sends a message to the *Item* object to check the available stock. If the item is in stock, it is added to the *OrderLine*.

The *Item* object sends a message to itself to check the reorder point on the item and, if it needs be reordered, sends a message to the *ReorderItem* to reorder the item then returns to *OrderLine*. If the item is not in stock, the *OrderLine* object sends a message to the order, placing it on hold.

EXAMPLE OF OBJECT-ORIENTED ANALYSIS: CREATE AN OMD FOR DOCUMENT SERVICES

Let us consider another type of service, one performed on business documents. When we work with files, for example, we perform file services. When we work with email messages, we request email services. We actually work with different applications built specifically for different types of documents. These applications have different visual interfaces and different sets of functions. The developers who created these applications began each time from user requirements limited by specific document types. Is this the right approach? Are there common features and properties for different documents, such as email messages and files? What is your answer?

CREATE THE DOCUMENTSERVICE MODEL

If you read the questions carefully, you can find the answer right there. The common word *documents* provides a base for common features and properties. Now we can look at this problem from a higher elevation and find more document types that can be modeled in the same way. We can think about forms, calendar and address book records, and linked articles in distributed knowledge bases.

By starting the design right from the beginning, we can save time on implementations for known documents, as well as bridges to new data types that may be unknown at this point. By consistently using the model design pattern, we can extract a conceptual part of the model into a base class or an interface. Let us discuss common features of different services. This discussion leads us to the *DocumentService* base class and the OMD for multiple types of data services presented in Fig. 1.18.

Data services work with documents or data elements. There are several operations that services perform on documents. We understand that operations can be very different, but again, we can look for common denominators. Generally speaking, every service *processes* documents. For some types of processing, data interpretation is needed. Let us add the *parseXML()* method to the base class. As I mentioned before, XML paves the way to unified services, and we will talk about this in more detail later. Specific services, such as a file manager or an email service, are subclasses derived from the *DocumentService* class.

The *DocumentService* class encapsulates common properties and behaviors of multiple document services. The subclasses that implement these specific services are much lighter now than they would be without this model. No repetition is necessary for common features implemented by the base *DocumentService* class.

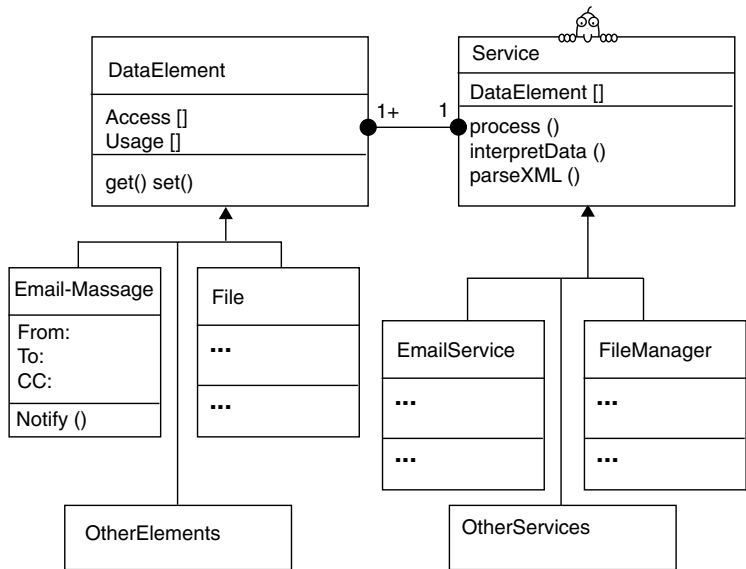


FIGURE 1.18. Document-Handling Services—Object Model Diagram.

Create the Document Model

A very similar model is created for multiple document types. The *DataElement* class represents the basic concept of a document. Every document type has its own data attributes. Common properties and operations can be found, based on an analysis of user requirements. Let us assume that data services are performed in a collaborative environment. In this case, documents can be grouped by security type to provide role-based access to them. For example, managers can access management-type documents, whereas administrators can access administration-type documents. Once we come to this conclusion, the security type becomes a common document property across all data types. We include this property and its associated methods in the *DataElement* class. This means that role-based access to different security type documents will be provided for files, email messages, and other known and unknown data types. Subclasses of the *DataElement*, such as *FileDocument*, *Email-Message*, and other data elements, inherit this ability without any extra code. Security values built into every data element provide for privilege-based secure data sharing.

A unified approach to handling multiple document types will lead to efficient reuse of functionality and (surprise!) to a consistent user interface throughout multiple document services. In Chapter 6, “Write Once,” you can find a more extensive discussion of document handling applications with more design patterns and code samples.

ARCHITECTURE STEPS: FIND PLAYGROUND-TIERS FOR YOUR OBJECTS

Divide each difficulty into as many parts as is feasible and necessary to resolve it.
—Rene Descartes (1596–1650)

What are the basic steps in architecture analysis?

- Define architecture goals and priorities.
- Decompose the system into presentation, services, data, and more layers.

- Define playground-tiers for application objects.
- Provide multiple architectural views

It is not necessary to finish your object-oriented analysis before starting to construct the architecture. On the contrary, these two processes work in parallel dimensions and have cross sections.

Enterprise applications in general are not single-user programs. The distributed computing nature of enterprise applications requires placing application objects on multiple tiers. It is possible that at this point, some objects can be redefined to avoid tight coupling and decrease network traffic.

Monolithic systems are easy to deploy but hard to change. Separating the presentation layer from the business logic is one of the most important design achievements.

We are going to undertake several steps of architectural analysis. We draw the lines between the presentation (client), business logic (server), and production layers (data and services) and find the right playgrounds (tiers) for application objects. At this point, you can expect some object redesign. This redesign can be related, for example, to the necessity to decrease the client object's dependency on server objects.

Before diving deep into the lake of architecture, let us take a high-level look. What is architecture? System architecture is a specialized view that focuses on a system's components and behaviors with no or minimum implementation details. Architects communicate this view in the form of design patterns, models, frameworks, and diagrams.

It is crucial to provide multiple architectural views. Look at any object from multiple sides and you can see more of it. Architectural views can be layered and can vary according to system specifics and viewer's goals:

- A set of Use Cases shows a **User's view** of the application.
- Object Model Diagram represents a **static structural view** of the system in the process of Object-Oriented Analysis.
- **Behavioral View** includes Sequence and Collaboration Diagrams
- Deployment Diagrams obviously communicate the way the application is going to be deployed or **Deployment View**.

For example, the deployment diagram will represent a solution for a problem we discussed before: what are the logical and physical tiers of the application? The art and science of architecture is to balance system capabilities, including performance, scalability, security, and other nonfunctional qualities in terms of the context of user requirements.

Unfortunately, it is not feasible to optimize all capabilities. For example, maximum performance can be achieved with minimum scalability and security. Architects need to define their architectural goals with clients and capture these nonfunctional capabilities with numbers and priorities as measurable items on the list of system requirements.

FROM SINGLE-USER TO CLIENT-SERVER AND MULTI-TIER ARCHITECTURE MODELS

In the early stage of the design process, we try to identify containers for the functions we want to provide. We can describe these containers in terms of layers and tiers. A single-user

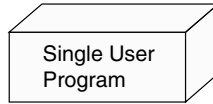


FIGURE 1.19. A Single-User Program—The Simplest Software Architecture.

program is an example of the simplest architecture (Fig. 1.19). It consists of a single tier—the process running the program.

Microsoft Paint can be considered an example of such a program. An IBM mainframe with old 3270 dumb terminals represents the next step to a multi-user system (Fig. 1.20). It is still a single-tier architecture. The programs run on a single tier—the IBM mainframe—while the terminals connected to the machine by long wires share no processing and justify their “dumb” alias.

A UNIX network can best represent the client–server architecture that was predominant in the 1980s. Client and server machines shared computer processing; in most cases, this sharing was provided by an X-windows operating system that separated presentation (keyboard, mouse, and display functionality) from computation, data storage, and networking, all of which were provided by a server/host computer (Fig. 1.21).

The client–server model failed to provide the necessary resources (e.g., database connections) to a growing number of clients. The work of servers was redistributed to business and data tiers. The business tier included connection pooling and other functions and mechanisms that provided scalability (Fig. 1.22).

Web technology brought another tier (or container) into the game. The Web container took responsibility for connectivity between multiple Web clients and services. The Web container includes a very efficient and inexpensive Web server (one of the best Web servers, Apache, is free) and/or an application server. The Web container and business processes can run on one server or be distributed on multiple machines.

The simplest glue that connects business processing with a Web server is the common gateway interface (CGI). The Web server intercepts client requests for program processing and sends them to executable programs. The program provides a necessary service—for example, retrieved or saved data—and in return generates a dynamic response in Hypertext Markup Language (HTML) format.

The Web server delivers this response back to the Web client, which is waiting for this response. The Web client, a Web browser running on a client machine, understands the HTML page and renders it into a context of text, graphics, and sound. The page regularly includes new links and/or forms that help the user navigate to more services.

The drawback of CGI technology, because of its simplicity, is that it has to create a new process for every client. When two clients request services—two programs—full-blown processes have to be created on the server. Ten clients hit the server, ten processes are

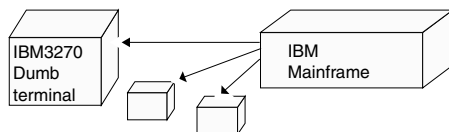


FIGURE 1.20. Multiple Users, Multiple Terminals, a Single Processor, and Time Sharing.

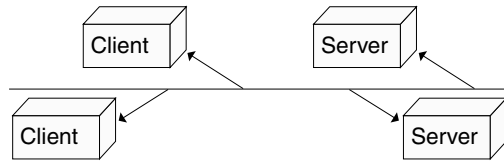


FIGURE 1.21. Client–Server.

created. A hundred clients hit the server—the server goes down. Simplicity and scalability are not always the best of friends.

Enterprise applications must serve thousands, sometimes millions, of clients. New technologies allow the Web server and business logic to be glued in such a way that every client request creates a new service thread on the server side. The thread is a very tiny process and much lighter than a regular one. This benefit multiplied a thousand times scales Web application capacity way up. One example of such an architecture is presented in Fig. 1.23. This example is based on J2EE components.

Tier 1. Client Container. Multiple types of clients request services via an *XML-based service application program interface (API)* using synchronous Hypertext Markup Transport Protocol (HTTP) or asynchronously using Java Message Service (JMS) or email with Simple Messaging Transport Protocol (SMTP) mechanisms.

Client types:

- Java Web Start or partner service application running on a corporate workstation (connects to Web container over HTTP or directly to JMS, JMail (SMTP), or application server in the service container using proper ports and protocols).
- Web (HTML) browser with or without a Java Applet (connects to the Web container over HTTP)
- Wireless devices, such as Smart Mobile Phone, personal digital assistant (PDA) with wireless application protocol (WAP), or iMODE-iApplix (a *defacto* standard established by NTT DoCoMo, a Japanese company).
- Wireless devices connected to the network with other radio frequency (RF) protocols, such as GSM, GPRS, 802.11 (WiFi), and Bluetooth.
- Operating environments for wireless devices range from WindowsCE and TabletPC (full Windows XP geared towards pen/voice interface) to PalmOS. These devices can run scaled down Java Virtual Machine versions, which are often presented under the single umbrella of the Java 2 Micro Edition (J2ME).
- Smart Cards, for example those based on JavaCard technology, secure personal identity features. Smart Cards require special devices—readers that used to be quite expensive in the past.

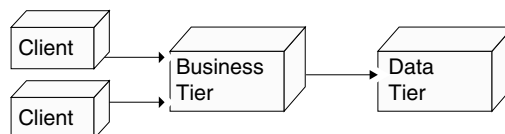


FIGURE 1.22. Multi-tier Architecture.

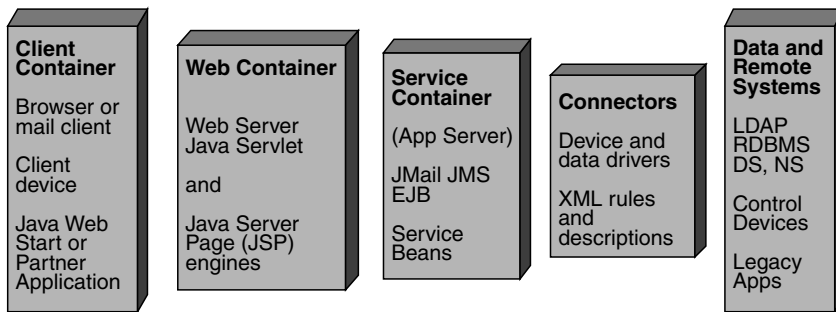


FIGURE 1.23. Example of the Architecture of a J2EE-Based Enterprise Web Application.

- Regular phones (connected to the Web container via a speech recognition system, for example one provided by Nuance.com)
- Email client programs (connect to JMail or SMTP server in the service container using SMTP)

Tier 2. Web Container. Web server, Java servlet, and JSP engines are glued into a single process in the Web container. The Java servlet engine is responsible for session tracking and request distribution, and JSPs provide a presentation layer back to the client.

Tier 3. Service Container. The service container can include generic services, such as Java Mail and JMS, and specific business services implemented with Java worker beans and classes. The services can be implemented as EJBs to gain the advantages of generic features like security and transaction monitoring provided by EJB vendors.

Tier 4. Connectors. This container includes the necessary software and hardware solutions that provide access from services to data and remote systems, legacy applications, and control devices. This book promotes a unified approach to data drivers and device controllers. Data drivers implement basic operations on data using Java Naming and Directory Interface (JNDI).

Data drivers are able to understand XML-based business rules and descriptors governing data access. The application business logic captured in the service container will never change, no matter what data source is used, whether Oracle RDBMS (relational database management system) or LDAP (Lightweight Directory Access Protocol). Device and system controllers implement a unified service model, which focuses on the interpretation function of controllers. The model considers the controller an interpreter with a set of possible input and output instructions. An important part of both models is an XML API and descriptors that capture system rules and behavior.

Tier 5. Data and Remote Systems. This container includes different types of data sources, legacy applications, production system terminals, remote systems, and devices controlled by the application or interacting with it. Data sources may include different types of RDBMS, LDAP, directory services, meta-directories, and so forth.

This example of multi-tiered architecture is one of many possible solutions. The architecture analysis will produce, among other design documents, a deployment diagram similar, or not very similar, to the one above. The deployment diagram will include more details. For example, we need to decide on communication protocols between tiers and provide

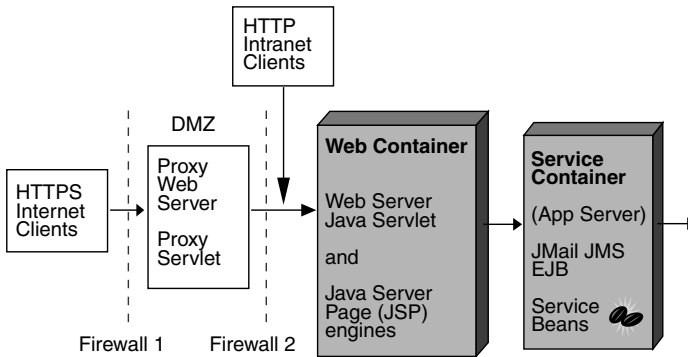


FIGURE 1.24. Demilitarized Zone (DMZ) in Web Applications.

security considerations. We need to consider client types expected by the application. The architectural solution will include presentation layers, or factories, for these types of clients. Choosing between thin and thick clients for a Web-based enterprise application is one of the more important architectural decisions.

The choices are related to your potential audience: for internal corporate clients, it is possible to control the Web browser type and version usage. This would allow, for example, reinforcing usage of the Java Plug-In for a Web browser to support the heavyweight Java Foundation Classes (JFC). Keep in mind that using JFC can easily add an extra megabyte or more to the applet size.

Java Web Start, a new client container, can deliver a full-blown Java application to your client over the Web and replace the Web browser as the most commonly used container. Unfortunately, Java Web Start-based applications will require significant resources on client machines. This can still work well for a selected group of Internet users. If your audience is the Internet population, it is a safe bet to use a Java applet supported by the major Web browsers, or to rely on HTML pages generated on the server side.

The user requirements might also include wireless access. Even if these requirements are not in place from the beginning, we can expect them to be in the future. We do not want to do extra work “just in case,” and we do not need to. We can solve this problem by separating business logic from the presentation layer. Placing the right objects on the right tiers is the key to this solution.

Communication between client and server containers can be provided with any protocol supported by Java: RMI, TCP/IP Sockets, or HTTP. This is valid for Intranet applications. Enterprise applications that are open to Internet users provide a security zone called the “demilitarized zone” (DMZ), between Internet clients and business services (Fig. 1.24).

The DMZ is based on two firewalls. Internet users can access generic Internet services, including a Web proxy, located inside the DMZ. The proxy servlet will redirect client requests to the Web container behind the second firewall. It is useful to remember that only the HTTP protocol is free to fly over the corporate firewall by default. For other protocols, special agreements with the security folks must be in place.

There is no firm line between architecture and the design process. Architecture is a general view, whereas design considers more details. It is possible that developers will need to reconsider their architectural plan during detailed design considerations.

BASIC DESIGN STEPS AND RULES

A repeatable design process requires discipline as well organizational and technological reinforcement to follow basic design rules:

- Focus on reusability: recognize reusable services and shared data.
- Use design patterns.
- Distinguish the conceptual part from implementation details.
- Balance overhead.
- Document design before coding.

It takes time and effort to acquire healthy habits for a proper level of documentation and communication in a team. The right approach to documentation sharing is important. Documentation should not die in files that are released once; it should have a living and easily updateable nature.

This book describes collaborative engineering technology that encourages privilege-based data sharing and increases members' feeling of ownership and willingness to be proactive. This technology keeps the team "on the same page" and minimizes the need for long and often unproductive meetings.

Recognize Reusable Services and Shared Data

Look for reusability from two points of view: within an application and throughout a multiapplication environment. For example, security services or data-caching mechanisms are good candidates for reuse. I have heard a lot of complaints about security services that require users to enter different names and passwords for different applications within a single company.

Of course, users at these companies may be under a lot of stress to remember several passwords, and their performance may suffer. Obviously, developers at their organizations did not think much about reusability when creating different applications. (Of course, they might be under stress, too.)

Reusability can be achieved only by extra work done on a specific part of the project. Then this part can be reused multiple times, providing a tremendous payoff. Focus on the idea that functionality can be reused and data can be shared. Two types of shared data should be recognized: data shared by different functions performed by a single user and data shared between multiple users.

For example, a single user's data are shared between different layers or screens within the application. This kind of data can serve as a common interface and should be stored, cached, on a tier that is the most active in processing this data. Each user/client works with its own set of the client's shared data. One of the mechanisms for caching client data is presented by the session object in Java servlet.

Data shared by many users, such as statistics or short database tables, can be stored on a proper tier as a unique copy that can be accessed by all the users running this application. One of the mechanisms to provide such access can be offered by static data.

Components of the user interface, or widgets (e.g., data fields and buttons), can also be considered a source of reusability. It is preferable to dynamically change widget state from visible to invisible, from active to inactive, or to change the color or label name instead of removing and recreating the widget.