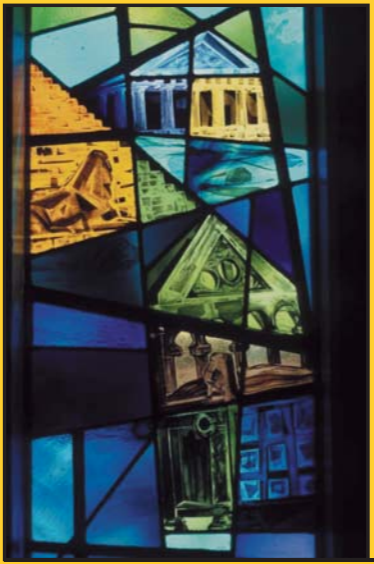


The J2EE™ Architect's Handbook

How to be a successful technical architect for J2EE™ applications

Derek C. Ashmore

Derek Ashmore has assembled a "must-have" book for anyone working with Java and/or J2EE applications. Mr. Ashmore covers all the bases in this "how-to" approach to designing, developing, testing, and implementing J2EE applications using Java with frequent references to XML, JDBC libraries, SOAP, relational database access (using SQL), and references various useful tools when relevant. This book clearly illustrates Derek's expertise in the Java world . . . thank you for sharing your knowledge to the IT community with such a useful book.

— Dan Hotka, Author, Instructor, Oracle Expert

[Derek has written] an in-depth and comprehensive resource for the Java 2 architect! The book provides a concise road map for real-world J2EE development. The approach is practical and straightforward, based on a wealth of experience. All aspects of project management, application and data design, and Java development are covered. This book avoids the "dry style" and over-abstraction (over-simplification) common to so many books in this subject area. An awesome book, I keep it on my "A" shelf!

— Jim Elliott, CTO, West Haven Systems, Inc.

Clear reading and bridges the gap between professionals and professors. I've read many technical books in my thirty-year career where the author spends more time tossing around the current buzz words and fails to get the point across. Derek's book really connects with the hard core developer. Practical, knowledgeable, excellent examples.

— John R. Mull, President, Systech Software Products, Inc.

I'm a junior J2EE technical architect, and I just finish reading your [book]. It's really interesting and instructive. It helps me a lot on my project planning . . .

— Bruno Gagnon, Junior Technical Architect

The J2EE™ Architect's Handbook

How to be a successful technical architect for J2EE™ applications

Derek C. Ashmore

What readers are saying about *The J2EE Architect's Handbook*:

I would highly recommend *The J2EE Architect's Handbook* to anyone who has struggled with how to practically apply all of the Objected Oriented Design, Design Pattern, eXtreme Programming, and Data Modeling books that line their shelves.

— D. Scott Wheeler, Partner, Open Foundation

This book is well crafted and explains everything you really need to know in order to be a successful and productive J2EE architect. It is an excellent book, which offers a full and detailed coverage of the topic of J2EE architecture and can be used as a handbook by the novice or as a reference tool for the experienced architect. The straightforward writing style and good visuals make for a quick and comprehensive learning experience. If Java is your primary programming language, and you're currently working as a J2EE architect or considering it as a future career, this book should be in your library.

— Ian Ellis, Senior Technical Architect

The J2EE Architect's Handbook is a must have for experienced architects and budding designers alike. It is concise, to the point, and packed with real-world code examples that reinforce each concept. Today's J2EE teams would do well to keep a copy at each designer's fingertips.

— Ross MacCharles, Lead Technical Architect

The *Architect's Handbook* offers an excellent summary/detail look at what comprises a mature J2EE application architecture. It helps the average developer to become a productive team member by providing an awareness of the larger issues in development. And it transforms the more senior technician into an insightful architect, now readily capable of making sound, high-impact decisions. An unparalleled resource for the architect's library.

— Jeffrey Hayes

DVT Press
34 Yorktown Center, PMB 400
Lombard, IL 60148
sales@dvtpress.com
http://www.dvtpress.com

ISBN: 0972954899

Library of Congress Cataloging-in-Publication Data
Ashmore, Derek C.
The J2EE architect's handbook : how to be a successful technical architect for J2EE applications / by Derek C. Ashmore.
p. cm.
Includes bibiliographical references and index.
ISBN 0972954899
1. Java (Computer program language) 2. Application software--Development. 3. Web site development. I. Title.
QA76.73.J38A84 2004 005.2'762
QB103-200953

Editor: Barbara McGowran
Cover Design: The Roberts Group
Interior Design: The Roberts Group
Indexer: The Roberts Group

Contents

Preface

The J2EE Architect's Handbook was written for technical architects and senior developers tasked with designing and leading the development of J2EE applications. With numerous strategies, guidelines, tips, tricks, and best practices, the book helps the architect navigate the entire development process, from analysis through application deployment. To help you achieve success as a J2EE technical architect, the book presents the following material:

- A basic framework for filling the role of technical architect
- Architect-level tips, tricks, and best practices
- Tips, tricks, and best practices for making code more maintainable
- Tips, tricks, and best practices for creating and communicating designs
- Out-of-the-box, open source support utilities for application architecture
- Template material for implementing many phases of application development
- Estimation and project-planning material

This book is *not* a study guide for any of the certification exams for Java and J2EE technologies provided by Sun Microsystems.

Further, the book is *not* for beginners. Readers should know Java syntax and have at least an intermediate programming skill set as well as basic knowledge of the following:

- ▲ Enterprise beans (experience coding at least one session bean and entity bean is helpful)
- ▲ Relational databases, SQL, and JDBC
- ▲ XML and how to access XML via Java
- ▲ JSPs and servlets
- ▲ Corporate systems development
- ▲ Object-oriented design concepts

A common misperception is that J2EE applications are incredibly complex. Authors of technical books and articles unintentionally support this fallacy by providing incredible technical depth on aspects of J2EE not commonly used. For example, many texts begin their discussions of enterprise beans by describing J2EE transaction capabilities in great detail. But most J2EE applications make only limited use of J2EE transaction management capabilities. In this book, I strip away some of the complexity that most developers rarely use to reveal how relatively straightforward J2EE applications really are. Your time is too valuable to waste reading about features and concepts you'll rarely use in the marketplace.

How the Book Is Organized

The first chapter of the book describes the role of the technical architect in most organizations and explains how the project life cycle illustrated in this book fits in with Extreme Programming (XP), the Rational Unified Process (RUP), and other possible methodologies.

Section 1 details how to define the project objectives using use-case analysis. It also discusses how to define scope and create a preliminary project plan. The guidelines presented in this section will help you successfully complete these tasks that are critical to your project coming in on time and on budget. The most common reasons for project failures or cost overruns are poorly defined and managed objectives and scope, not technical problems.

Section 2 focuses on object-modeling and data-modeling activities, describing how detailed they need to be and illustrating common mistakes. In addition, you will learn how to architect interfaces with external systems and how to refine the project plan and associated estimates. The modeling skills presented in this section are critical to effectively communicating a design to developers.

Section 3 presents implementation tips and guidelines for all aspects of J2EE applications. You'll learn how to layer your application to minimize

the impact of enhancements and changes. You'll also become acquainted with CementJ, an open source assistance framework that enables you to streamline your development at each layer.

In addition, section 3 details application architecture decisions you'll need to make regarding testing, exception handling, logging, and threading, and you'll learn tips and techniques for implementing major sections of the design. The failure of a technical architect to define implementation strategies and methodologies can slow down a project significantly and increase the number of bugs.

Section 4 offers tips and guidelines for developing testing procedures and process improvement. These suggestions are directed at making your applications more stable and maintainable. In addition, you'll learn the signs warning you that refactoring activities are necessary. Reading this section will enable you to make subsequent projects even more successful.

Common Resources

The book utilizes the CementJ open source project. A Java API, CementJ provides the functionality that most J2EE applications need but is not yet directly provided by the JDK specification. CementJ helps you build a strong foundation for your application, filling the gaps between the JDK and your applications. Using CementJ will help you streamline your development. CementJ binaries and source can be downloaded at http://sourceforge.net/projects/cementj/.

Another open source project on which this book relies is ProjectTrak. This planning package serves as an illustration for implementing the concepts presented in this book. ProjectTrak binaries and source can be downloaded at http://sourceforge.net/projects/projecttrak/.

Errata, example source code, and other materials related to this book can be found at http://www.dvtpress.com/javaarch/.

Feedback

I'm always interested in reading comments and suggestions that will improve future editions of this book. Please send feedback directly to me at dashmore@dvt.com. If your comment or suggestion is the first of its kind and is used in the next edition, I'll gladly send you an autographed copy.

Acknowledgments

Several people helped tremendously to refine the book concept and edit the drafts to keep me from mass marketing mediocrity. They have my undying

gratitude and thanks. I could not have written this book without the assistance of the following people:

Ron Clapman is a seasoned and experienced senior architect in the Chicago-land area. Ron started his career at AT&T Bell Laboratories in the 1980s where he received firsthand knowledge on the then new and growing field of object-oriented software. Today, he provides a broad range of services that encompass multiple roles as technical project manager, business analyst, enterprise application architect, and software developer for mission-critical applications and systems. Ron's reputation as an architect, lecturer, teacher, and mentor are highly recognized and valued by his clients.

Jeff Hayes is an independent application software engineer with a background in mathematics, financial and medical systems application development, and bioelectric signal processing. He has a diverse list of clients whose primary businesses include nuclear power engineering, medical facilities quality assurance, hospitals, and banking systems and trust management. As owner of Chicago Software Workshop, Jeff stresses education and long-range planning in his engagements. He holds a master of science degree in electrical engineering from Northwestern University and is a member of the IEEE.

Ross MacCharles is the lead technical architect for Nakina Systems in Ottawa, Canada. He has spent his fourteen-year career as an influential solutions provider and technology leader in a variety of business domains, including global telecommunications firms, innovative startups, federal government agencies, international news agencies, major banks, and insurance companies. Ross can be reached at rossmacc@hotmail.com.

Jason Prizeman is a technical architect and has been specializing in the J2EE framework for more than five years. His project exposure is vast, encompassing a huge range of projects for an array of business sectors.

Mike Trocchio is a senior technical architect for Leading Architectures Inc., a Chicago-area consulting firm. Mike is currently focusing on the design, development, and deployment of large-scale Internet applications. He has strong expertise in all areas of information technology, including requirements gathering, object modeling, object-oriented designs, database modeling, code development, implementation, and performance tuning. He also has strong exposure to application security, general corporate security policies, and using open source products in an application to save time and money. Mike can be reached at mtrocchio@leadingarch.com.

D. Scott Wheeler has performed almost every role in systems development over the past sixteen years, while utilizing a wide variety of

technologies. He is currently a technical architect involved in open source development and promotion. He is the founder of the Open Source Developer's Kit (http://www.osdk.com/), owner of Nortoc Inc. (http://www.nortoc.com/), and partner in Open Foundation (http://www.openfoundation.com/). Scott can be reached at dwheeler@nortoc.com.

1

Project Development Team and Project Life Cycle

This chapter lays the foundation for building a successful first project, from inception to release. It begins by defining what a technical architect is and does and summarizes how the architect works with other team members. The chapter continues with a look at a few alternative approaches to the development process. Still a subject of considerable debate, the definitive process for building a successful project does not yet exist, leading many companies to adopt a hybrid plan.

Project Development Team: Roles and Responsibilities

All J2EE development teams need people with a wide variety of skills to fill numerous roles within the team. Among the many skill sets needed to make a J2EE project successful are:

- ▲ Technical architect
- ▲ Project manager
- ▲ Business analyst
- ▲ Layout designer
- ▲ Presentation-tier developer

- ▲ Business logic developer
- ▲ Data modeler
- ▲ Database administrator
- ▲ Data migration specialist
- ▲ Infrastructure specialist
- ▲ Testing specialist

Although the book focuses on the role of the technical architect, this section defines the roles and responsibilities of other major players on the J2EE development team and describes the responsibilities of the technical architect with respect to those roles.

Some organizations use different labels for the roles. For instance, an infrastructure specialist may be called a system administrator, a testing specialist may be a tester, and some organizations may distinguish between a test team manager and individual testers. Regardless of the terms you attach to these skill sets, making all of them part of a development team greatly increases its chances of creating a successful J2EE project.

Further, it's possible for one person on the team to fill many roles and for one role to be co-owned by multiple people if the project is large enough. Some organizations combine the roles of technical architect and project manager. Some organizations have a senior developer double as a database administrator or as an infrastructure specialist. And some have the same developers work on the presentation tier as well as the business layer. I'm not trying to recommend a team organization but merely to communicate what skill sets are necessary, however they are organized.

Technical Architect

The technical architect identifies the technologies that will be used for the project. In many organizations, some technology choices are made at an enterprise level. For instance, many organizations make hardware and operating system choices and some software choices (e.g., the J2EE container vendor) at an enterprise level. Commonly, choosing a language, such as Java, is an enterprise-level decision.

However, most applications have technical requirements that aren't explicitly provided in enterprise-level edicts. I make a distinction between technology choices made at an enterprise level and those made for individual applications. For example, a decision to use the Java language for all server-side programming might be made at an enterprise level, but a decision about

which XML parser to use might be left to individual application architects. In many organizations, the people making enterprise-level technology choices make up a group separate from the J2EE development team.

The technical architect is commonly responsible for identifying third-party packages or utilities that will be used for a project. For example, the architect might identify a need for a template-based generation engine and choose Apache's Velocity.

The technical architect recommends the development methodologies and frameworks of the project. Typically, the architect makes these recommendations to the project manager. For example, a common recommendation is to document all analyses in use-case format and supplement with a prototype. Another common recommendation is to document the design in terms of an object model. Some organizations define the methodologies used at an enterprise level.

The technical architect provides the overall design and structure of the application. Each developer brings to a project a unique set of preconceived opinions, habits, and preferences. Synthesizing the input of this sometimes widely divergent group, the technical architect ensures that the work done by individual developers is complementary.

I liken the role of technical architect to that of an orchestra conductor. All musicians have differing opinions about how to interpret a given work. The conductor provides the interpretation that will be used and works with the musicians to implement it.

The technical architect ensures that the project is adequately defined. A project analysis must be detailed and consistent enough to form the basis for building an application. Typically, the technical architect works with the project manager and business analyst to define the project.

The technical architect ensures that the application design is adequately documented. Documenting the application design is a critical step in establishing sound communication with and among developers. The process of creating documentation forces the architect to think through issues thoroughly. And the final document enables management to add or change developers to the project without adversely encroaching on the architect's time. For developers, documentation allows them to proceed if the technical architect is absent from the project for a limited period and enables them to work through design inconsistencies on their own without consuming the

time of other members of the development team. Documentation also helps to insulate the project against the effects of personnel turnover.

I've seen many projects that were not documented, and the result was that adding a developer was a major chore because the architect had to verbally convey the design to the newcomer. Having to communicate the design verbally negates some of the benefits to bringing on additional developers.

The technical architect establishes coding guidelines. Because individual developers have coding preferences, coding standards need to be articulated so that the individual pieces are more easily brought together. The technical architect is responsible for establishing project procedures and guidelines for topics such as the following, which are covered in more depth later in the book:

- ▲ Exception handling
- ▲ Logging
- ▲ Testing
- ▲ Threading

The technical architect identifies implementation tasks for the project manager. This role is especially important for J2EE projects because they encompass a much wider range of technologies than do other types of systems projects. Out of practical necessity, the technical architect also helps the project manager with project planning and estimates.

The technical architect mentors developers for difficult tasks. Typically, the architect is more experienced than the developers. When the developers run into a technical problem that slows them down, the architect is often the one to help them create a solution. For many projects, the architect is more of a mentor than an implementer.

The technical architect enforces compliance with coding guidelines. Being the one who establishes coding guidelines, the technical architect is the most likely to recognize when the guidelines are not being followed and is therefore the logical choice to enforce them. A project manager, who typically is charged with enforcement tasks, often does not have the technical experience to recognize compliance.

Code reviews are an excellent enforcement mechanism. It is much harder for individual developers to privately skirt team coding standards if other team members examine the code.

Code reviews are also an excellent learning tool for all members of the development team. The technical architect discovers holes in the design, and all participants learn tips and tricks from the rest of the team. Typically the most experienced member of the team, the technical architect often facilitates the code review. To be most useful, a code review should be held in a congenial, nonthreatening atmosphere.

The technical architect assists the project manager in estimating project costs and benefits for management. Although this is usually the project manager's responsibility, most project managers are less experienced with J2EE technologies and may not be aware of everything that needs to be done.

The technical architect assists management in making personnel decisions for developer positions. While personnel decisions are often viewed as a management function, the technical architect is in a good position to assess technical competence. Mistakes in personnel decisions can cause considerable damage to project timelines.

Project Manager

The project manager is responsible for coordinating and scheduling all tasks for all members of the project development team. The project manager must also communicate current project activities and status to management and end-user representatives. Further, the project manager acquires any resources or materials needed by the project or the team members.

The technical architect is responsible for providing technical advice and guidance to the project manager. The technical architect assists the project manager in identifying project tasks and the order in which they should be completed. The architect also helps the project manager identify needed materials and resources, including guiding the selection of other team members and validating their skill sets from a technical standpoint.

Business Analyst

The business analyst is responsible for working with end users to define the application requirements—the detail necessary to design and build the application. Because end users and developers often use different terminology, the business analyst is responsible for translating communications between end users and developers. Often the business analyst has experience on both the end-user side of the enterprise and the information technology side.

As a project progresses, the business analyst's role diminishes but does not disappear. Developers typically have additional business questions that come to light during coding and testing activities. The business analyst works with the business side to get these questions answered.

The technical architect is responsible for ensuring that the application requirements determined by the business analyst are adequate. It's unreasonable to expect 100 percent of the analysis to be complete and correct. After all, analysis is to some extent subjective. However, the analysis needs to be complete enough to warrant proceeding with design.

Layout Designer

Many applications, especially those that are publicly available, need professional graphics or layout designers. Most technical architects, left to their own devices, can produce functional Web pages, but those pages typically are ugly and hard to use. Graphics design is more art than science. Usually, the layout designer works primarily with the business analyst and other representatives of the business side to work out the design. But the layout designer may also work with the presentation-tier developer to create a prototype.

The technical architect is responsible for ensuring that the layout is technically feasible. I've seen many Web page designs that use text effects that are available in word processors but are not supported by HTML—for example, a design using text rotated 90 degrees. The architect is in a position to catch and correct these kinds of problems early.

Presentation-Tier Developer

The presentation-tier developer is responsible for coding all HTML, Javascript, applet/Swing code, JSPs, and/or servlets for an application. In general, anything directly involved in producing the user interface is in the purview of the presentation-tier developer. Typically in collaboration with the layout designer, the presentation-tier developer builds the prototype and develops the working version. And with the technical architect, the presentation-tier developer determines the structure and design of front-end navigation.

The technical architect is responsible for ensuring that design patterns can be maintained and extended. Navigation issues are often complex and can easily degrade into hard-to-maintain code. The technical architect is in

a good position to identify and correct maintenance issues as well as other technical problems that arise.

Business Logic Developer

The business logic developer is responsible for coding all invisible parts of the application, including enterprise beans, Web services, RMI services, CORBA services, business objects, and data access objects. Some people refer to these invisible parts as the server-side components of the application. The business logic developer is often a Java specialist who works closely with the technical architect and assists in performance tuning as needed.

The technical architect provides guidance for the business logic developer. It's common for technical issues and problems to arise in server-side components, which are usually the most complex pieces of an application. Thus the technical architect often acts as a mentor to the business logic developer.

Data Modeler

The data modeler uses information from the business analyst to identify, define, and catalog all data the application stores in a database. Data modeling typically involves documenting application data in entity-relationship (ER) diagrams. The database administrators then uses the ER diagrams to produce a physical database design. Thus it is common for the roles of data modeler and database administrator to be combined.

The technical architect is responsible for ensuring that the data model is adequate. As with business analysis, it's unreasonable to expect the data model to be 100 percent complete. If the data model is largely complete and in third normal form, future changes in the model (and thus the database) are likely to be minor.

Database Administrator

The database administrator is responsible for formulating a database design based on the business requirements for the application and for creating and maintaining database environments for the application. Typically, the database administrator assists with performance tuning and helps the business logic developer diagnose application development issues regarding data access. Sometimes, the database administrator doubles as a business logic developer or data migration specialist.

The technical architect works with the database administrator to resolve any issues or problems involving database storage. However, the database administrator primarily interacts with the data modeler and the business logic developer.

Data Migration Specialist

Some applications, such as those for data warehousing, depend heavily on data migrated from other sources. The data migration specialist writes and manages all scripts and programs needed to populate the application databases on an ongoing basis. When an application has few migration requirements, this role may not be necessary or may merge with the database administrator's role.

The technical architect defines data migration requirements for the migration specialist. Working with the data migration specialist to solve any technical issues or problems that might arise is another aspect of the technical architect's role.

Infrastructure Specialist

The infrastructure specialist is responsible for providing all development, testing, and production environments as well as the deployment methods. A formal infrastructure for development and deployment saves time and effort. The idiosyncrasies involved in administrating containers, writing deployment scripts, and assisting with other developers diagnosing problems with their test environments represent a unique and challenging problem set.

The technical architect defines infrastructure requirements for the infrastructure specialist. The architect works with the specialist to determine the number and nature of the environments needed and what level of support is required for each environment. Many projects need at least one development, testing, and production environment. Some organizations combine the role of infrastructure specialist with that of technical architect.

Testing Specialist

A testing specialist is typically a detail-oriented person who makes sure that the application produced matches the specification and is reasonably free of bugs. Typically, a testing specialist has at least a basic knowledge of the business area.

The technical architect works with testing staff to identify any infrastructure requirements and support needed. The project manager and the business analyst usually establish the content of test plans and the testing methodology. Therefore, the architect's role in testing is usually support.

Project Life Cycle Approaches

There are differing schools of thought as to what the J2EE project life cycle should be. This section describes these schools of thought and presents my views on the topic. The guidelines presented in this book are intended to be compatible with any methodology.

Waterfall Approach

The waterfall approach entails performing all analysis and design for a project before coding and testing. This approach was commonly used when most development was mainframe-based and is still the one most companies prefer.

Projects developed under the waterfall approach tend to be large and have long delivery times. Hence, they entail more risk. These projects usually don't require business participants to learn as much technical terminology, and the business-side interface is tightly controlled.

Compared with other approaches, the waterfall approach to project development does not provide feedback as early in the process but delivers a more complete solution. Waterfall projects tend to fit neatly into the budget planning cycle, which may be one reason for their popularity.

Because of the length of time waterfall projects usually require, the business requirements often change during the project. Project managers then face a dilemma: if the project doesn't change with the business, the resulting application won't provide as much benefit; and if the project changes course to follow business requirement changes, the time and resources needed for the project will be negatively affected.

Iterative Approaches

Iterative approaches strive to separate a project into small component pieces that typically need few resources. Thus the iterative approach is the antithesis of the waterfall approach. The most popular iterative method is Extreme Programming (XP).

The central objective of XP is reducing the technical risks and project costs that plague the waterfall approach. XP uses the following assumptions:

- ▲ Catching mistakes earlier is cheaper in the long run.
- ▲ Reducing complexity also reduces technical risk and is cheaper in the long run.

XP dictates that you break the problem up into many small problems (called stories) that take three weeks or less to implement. Each story is co-developed by two programmers using one machine. The programmatic test to determine if the new story functionality works is developed and added to the regression test suite when the story is developed. These programmers ignore every aspect of the application except the story they are working on. A business participant is dedicated to the project and is immediately available to answer any business questions that arise.

Using pairs of programmers to code everything theoretically reduces the probability that an error survives to deployment. Using pairs also tends to make code simpler because it takes time to explain the concept to another person. The more complicated the algorithm, the harder it is to explain. The emphasis on reducing complexity makes it less likely that mistakes will occur.

The emphasis on testing, creating, and frequently running a regression test suite catches mistakes early and reduces the probability that any change will inadvertently introduce new bugs or have other unintended consequences.

XP reduces risk by providing feedback early. A development team proceeding down the wrong track will be alerted and corrected earlier, when it's much cheaper.

Rational Unified Process

The Rational Unified Process (RUP) is a formalized development methodology. Most RUP literature describes it as an iterative approach, but that's only half the story. RUP emphasizes starting with requirements gathering, analysis, and design activities for the entire project—including object and data modeling—before proceeding to construction. In this sense, RUP takes a waterfall approach to analysis and design but an iterative approach to construction and delivery. By encouraging early requirements gathering and analysis, RUP seeks to keep the project aligned with user expectations.

RUP mitigates risk by encouraging the team to develop the riskiest portions of the project first, allowing more time to recognize and respond to issues and problems. It also reduces rework when the design requires alteration.

Which Approach Is Most Popular?

I'm not prepared to declare any of these approaches "best." They all have advantages and disadvantages. The waterfall approach appears to be the most commonly used.

XP is rarely used in pure form. This isn't a judgment, merely an observation. Most companies use (and are more comfortable with) a waterfall approach to initial development and major enhancements. While enhancements are more iterative with the waterfall approach, the iteration size is usually much larger than with XP approaches.

XP's requirement of two coders for one task is a hard sell. From a layman's perspective, XP appears to consume more resources than is necessary for any given coding task. The cost for the extra coder is readily quantifiable, but the lost productivity for mistakes often is not. And people tend to opt for reducing the costs they can easily see, not the ones they may know are present but are not apparent.

RUP seems to be gaining popularity. In fact, more companies appear to be using RUP than are using XP. However, every implementation of RUP I've seen has been partial. It's common for organizations to be selective and use the portions of RUP that provide the most benefit to the project at hand.

Consider a Hybrid Approach

This book is largely compatible with either approach. XP users would merely choose much smaller iteration sizes than my illustrations. Because one approach rarely has a monopoly on common sense and is devoid of disadvantages, I prefer a hybrid approach.

XP's emphasis on testing has great value. I've adopted the practice of coding test sequences for everything I write and combining them into a full regression test. I've even seen a team go so far as to put a full regression test in the build and force the deployment to fail if all the tests don't pass. I find that the mistakes avoided by this practice more than pay for the extra time and effort required to develop and maintain test scenarios.

XP's war on complexity has value. Simpler is better. Ignoring all stories but the one you're working on does produce simpler code in the short term. But it also introduces a higher probability of rework (or refactoring, in more modern parlance), for which many projects have no budget. If refactoring isn't done properly or the developers are under time pressure, the code can easily end up being unnecessarily complex anyway. Also, many developers use the "complexity" excuse to ignore business requirements.

RUP's emphasis on centralized analysis and design has great value. XP assumes that developers can take a parochial view of the story they are working on and ignore anything else. This can cause some amount of rework. All developers really should have a larger focus. Because RUP concentrates analysis and design at the beginning of a project, it represents a sensible compromise between a purely iterative approach and the waterfall approach.

It is necessary to control communication with end users. XP assumes that any member of the development team should be able to talk to an end-user representative. Developers and end users usually have different perspectives and use different terminology. In practice, many developers have trouble adapting to nontechnical terminology. They simply can't translate business terminology into technical terminology, and vice versa. Some centralization of communication to the business side is necessary as a practical matter.

Further Reading

Beck, Kent. 2000. *Extreme Programming Explained.* Reading, MA: Addison-Wesley.

Brooks, Frederick P., Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley.

Kroll, Per, and Philippe Krutchen. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Boston: Addison-Wesley.

Section 1

Planning J2EE Applications

The technical architect typically assists in planning J2EE applications by participating in analysis activities, defining scope, and estimating resources, among other activities. The architect's role in the planning stage varies greatly from company to company. Although I've taken the perspective that the architect leads and facilitates planning activities, your role at an individual company may be to assist rather than facilitate.

In this section, you will learn how to:

- ▲ Facilitate and document business analysis.
- ▲ Assist the project manager in defining project scope.
- ▲ Estimate needed time and resources.
- ▲ Define and design interfaces to external applications.

The skills you learn here will enable you to apply the design techniques discussed in section 2.

2

Defining the Project

The first step in developing any application is performing analysis to define its purpose, scope, and objectives. A J2EE application is no exception. Including analysis in the development process is basic common sense, but I'm continually amazed at how many projects muddle through without defining their targets first.

The technical architect is not directly involved in defining the project; that is the job of the project manager, business analyst, and end user. However, the architect *is* responsible for ensuring that the project is defined with enough consistency and detail that it can be physically designed and implemented. Because most other members of the J2EE development team don't know what information is required to design and implement an application, the technical architect often facilitates project definition discussions.

The technical architect must possess analysis skills. Without analysis skills, an architect cannot recognize weak points and gaps in project definition. Although an architect who lacks analysis skills will likely catch most problems in project definition during the construction phase, by then it's more expensive to make changes.

I can hear the groans of developers as I write. Technical people want to hear more about coding techniques than project definition and analysis-gathering strategies. I completely understand. There's nothing I like more

than producing good code that does something useful. However, to get good code, you need good analysis and good project definition. My experience is that the probability of getting useful code without doing decent analysis first is slim to none.

Use cases are an important tool in analysis. The Unified Modeling Language (UML) specification was created to describe and document analysis and designs for systems that use object-oriented languages such as Java. The main construct UML has for describing what an application will accomplish is the *use case*. This chapter defines the term *use case*, guides you through writing use cases for a project, lists some common mistakes made in creating use cases and how to avoid them, and presents and discusses an example of use cases written for one project.

This chapter does not contain a comprehensive synopsis of use cases with respect to the UML specification. I present the subset of the specification that is commonly used and is practical. For a thorough treatment of the UML specification, see Booch, Rumbaugh, and Jacobson (1999).

Although some developers distinguish between use cases and requirements, I see no difference. Requirements are the specific features, written in business terms, that an application must provide. Therefore, requirements typically are use cases written in summary form.

If you're using Extreme Programming (XP), you create stories rather than use cases. Nonetheless, you will still find this chapter useful. Despite the more granular nature of the stories most XP users create compared with UML use cases, I consider the constructs of the story and the use case to be conceptually identical. The major difference is that the granularity of the story enables one pair of programmers to implement it in three weeks' time versus the larger time frame usually needed for one programmer to implement a use case.

Additionally, I like to prototype the user interfaces. A prototype is an excellent vehicle for enabling the business side and developers to understand the target of a development project. I usually have no trouble getting the business side interested in the prototyping process because it concretely represents what they're going to get. Prototyping also helps refine the use cases.

Once you've defined a project's use cases (or stories), you can create a fairly detailed definition of the project, written in business terms, that both developers and businesspeople can understand. This allows the business side and any management stakeholders to provide feedback early. Getting a formal

sign-off for the use cases in a particular project enables the project manager to contain project scope.

Identifying Project Scope

A high-level project definition and preliminary idea about scope is needed before use-case analysis and prototyping exercises can be effective. Most developers are detail oriented and will consider this high-level definition too vague to be useful. Keep in mind that the purpose of this high-level project definition is only to determine scope for the use-case analysis (not for coding).

Here is an example of a high-level project definition statement: Build a system that assists project managers in planning tasks, tracking the activities of every team member, and estimating completion dates. The project-tracking application should allow the user to do the following:

- ▲ Define a project and its tasks.
- ▲ Record people assigned to project tasks and estimate the time needed to complete each task.
- ▲ Record the order in which tasks will be completed.
- ▲ Record project progress and mark tasks as completed.
- ▲ Automatically schedule the project.
- ▲ Create reports of project progress.

As vague and simplistic as the statement is, it provides a starting point for identifying actors and constructing use cases.

Identifying the Actors

The first step in use-case analysis is to identify the actors. An *actor* is the user type or external system serviced or affected by the use case. Although word *actor* has connotations of being an actual person, a UML actor can be an external system or an organization type or role.

The following is a list of actors for a report generation application:

- ▲ Trust customer user
- ▲ Trust customer organization
- ▲ Banking support user
- ▲ Report template developer
- ▲ Document delivery application interface

- ▲ Report template definition interface
- ▲ Data warehouse application interface
- ▲ Report request application interface
- ▲ Application administrator

And here's a list of actors for a cash-tracking application:

- ▲ Cash manager user
- ▲ Transaction approver user
- ▲ Senior transaction approver user
- ▲ Banking support user
- ▲ Fund accounting application interface
- ▲ Application administrator

You may have noticed that in each example, I listed individual user groups as separate items. I did this primarily because every user group has different capabilities. While some use cases apply to all types of end users, others are user-group specific. In each example, we had far fewer different types of end users when we started use-case analysis than we had when we finished. During the course of writing the use cases, we began to realize that there were different user roles that required different capabilities.

It is possible for a user to represent multiple actors. For example, a user who provides banking support may also assume the role of a cash manager or transaction approver.

Consider the application administrator as an actor for any large application. This forces some attention to support—which increases availability and in turn makes other actors (who happen to be people) happy.

Make sure that all actors are direct. Sometimes people are confused by the external system interfaces and want to list as actors the end users serviced by an external interface. For example, if a security-trading system is one of the external interfaces, you may be tempted to list traders as actors because they are serviced indirectly by your application. However, the security-trading system is the actor, and the traders are indirect end users.

Facilitate identifying actors by beginning with a small group. Technical architects and business analysts can facilitate the discussion by making assumptions about who the actors are and reviewing them with other members

of the team and the business side. In my experience, people are much better and quicker critiquing something in place than they are adding to a blank sheet of paper. You will probably discover additional actors as use-case analysis proceeds.

Writing Use Cases

A *use case* is a description of something a system does at the request of or in response to an action by one of its actors. You should write use cases in business terms, not technical ones. Anyone on the business side should be able to read the text without a translator or technical glossary. Use cases containing technical terms often indicate that technical design assumptions are being made at this stage, and they shouldn't be. Use cases can also serve as a casual "contract" between the business and development sides of the organization as to what will be delivered in what increments.

Use-case text should begin "The system (or application) will." If you identify a use case that cannot be written in this form, it's likely not a valid use case but part of another one. Note that use cases often service multiple actors. I recommend explicitly listing all affected actors in the use case.

The following are examples of use cases from a reporting system:

- The system will provide an interface that will accept report template definitions from an existing MVS/CICS application.
- The system will allow application administrators to control the report templates that members of a trust customer organization can run.
- The system will run reports at least as fast as its predecessor system did on average.
- The system will restrict reported data for all trust customer users to that of the trust customer organization to which they belong.
- The system will allow banking support customers to execute all report templates using data from any trust customer organization.

Some of these use cases have additional detail beyond the summary sentences. For example, complete use-case text for the performance requirement is:

- The system will run reports at least as fast as its predecessor system did on average. Trust customer users and banking support users run reports. The primary measurement is the clock time measured from

the time the submit button is pressed until the time the user is able to view the report in the browser. CPU time is not relevant to this use case. Performance and scalability were the entire reason the rewrite project was funded.

Uses cases can be written with a more formal organization and content. See Cockburn (2001) for more details.

There are no rules about how long a use case should be. Generally, more information is better. I find it helpful to start with and include a summary for each use case that is no longer than two sentences. This simplifies organizing the use cases as the list grows. As analysis proceeds, you will attach additional detail to most use cases.

Avoid use-case diagrams. The UML specification does define a graphical representation scheme for use cases. However, graphical schemes are rarely used, and I purposely do not discuss them in this book. My experience has shown that use-case diagrams confuse both the business side and developers, and that the costs of creating, explaining, and maintaining these graphical constructs far outweigh any benefits they provide.

Writing use cases requires in-depth participation from the business side. From the technical side, some business analysts may be able to help construct an initial draft, but the process should not end without direct business side participation and review. Although enlisting the involvement of business users is sometimes easier said than done, their input is valuable. In my experience, insufficient business support for analysis efforts such as use-case review can cause a project to fail.

Facilitate use-case analysis by starting with a small group. Technical architects can speed this process along by working with one business side user or a business analyst to draft a set of use cases that can initiate discussion. These draft use cases will be incomplete, and some will be incorrect, but you'll get feedback easier and quicker than you would if you started with a blank sheet of paper. You can use objections to your assumptions to refine and improve the draft use cases.

Consider recording use cases in a database. I find it helpful to enter the use cases into a database rather than using a word processor. Please see the "Use Case Template Database" (defined using Microsoft Access) on the Internet at http://www.dvtpress.com/javaarch/.

Enlist someone to act as "scribe" for the use-case discussions. When you're facilitating a discussion, you won't have time to take good notes. Having someone other than the facilitator write the discussion notes helps ensure that they will be complete and understandable.

Write use cases so they can be amended as more information becomes available. Use cases are always evolving. If you discover additional information in the modeling phases or in later portions of the project, add this material to the use cases.

Use-case analysis is finished when team members feel they can estimate a time to implement each use case. Estimates may be in terms of number of weeks rather than hours. Some developers don't feel comfortable providing estimates until they've essentially coded the application. You may need to gently remind these developers that some difference between the estimate and the actual amount of time a task takes is expected.

Be sure to include requirements for security, scalability, and availability. The following are use cases for these three topics from systems I've architected in the past:

- The system will require senior approver users to approve cash transactions exceeding $5 million.
- The system will require separating the transaction entry user and the approver.
- The system will have reasonable response times for all users with at least eighty concurrently running reports.
- The system will be available 24x7x365 with the exception of a fifteen-minute maintenance window on Thursdays at 10 p.m., provided that Thursday is not within five business days of month-end.

Do not slow down if the group has trouble articulating requirements. Make assumptions and proceed. If your use cases are not right, the objectors have the responsibility to tell you what's wrong so you can correct the problem. You can use that information to refine and improve the use cases.

Common Mistakes

This section contains examples of use cases that have various defects.

Imposing a technical design assumption under the guise of a requirement. This is the mistake I see most frequently. Consider the following use case paraphrased from the reporting system example used earlier in the chapter:

▲ The system will allow application administrators to limit system load by setting rules that prohibit report execution for groups of users or redirect their execution to a batch stream.

This use case made several unwarranted assumptions and had us solving the wrong problems. It assumed that the hardware/software architecture used by the application could not be scaled to handle the load and that some alternative processing route was necessary. It assumed that the application could not be made as efficient as the "batch stream" mentioned. And it assumed that the batch stream environment in fact had surplus capacity to handle the load that the application should have been handling.

Even if some of the assumptions made in this use case turned out to be true, we should have started by planning an architecture that more than supported our load. In fact, most of the assumptions turned out to be false: The architecture could handle the load efficiently; the batch stream was a frequent performance bottleneck and, in fact, did not have surplus capacity; and the efficiency of the application more than satisfied users.

A better way to write this use case would have been:

▲ The system will support up to 200 concurrently running reports with a maximum daily volume of 500,000 reports.

Including physical design assumptions in use cases. For example, one of the developers submitted the following use case for the reporting system:

▲ The system will insert a row into the report request table after the request is completed.

This use case made the physical design assumption that we were recording request runs in a table. But at that point, we had not decided whether we would or wouldn't do so, nor should we have. After some discussion, I learned that application administrators needed a way to know what reports a user ran so they could reproduce problems a trust customer reported to the help desk. Given these requirements, a better way to word the use case would have been:

- ▲ The system will record report request history for at least thirty-six hours for the benefit of application administrators and report template developers investigating reported problems.

Not keeping analysis sessions productive. Analysis efforts can stall for many reasons, including ineffective facilitation or leadership, an extremely low level of detail in the discussion, and lack of information. Technical architects can steer the development team away from all these problems.

Failing to document use cases, even when the project is small. Most developers assume that documenting use cases is unnecessary when the project has only one developer. Use cases should be documented anyway. Documented use cases target the development effort and make tangents less likely. Further, documented use cases communicate the objectives of the development to management and the business side and assist in project transition if additional developers join the team or the project is assigned to another developer.

Repeatedly defining terms in every use case. Even for complicated applications, it's unnecessary to define terms repeatedly in every use case in which they appear. Instead, you can define and maintain them once in a separate list of business terms. For example, *cash transaction* (a term used in an earlier use-case example) refers to money transferred from one account to another. The characteristics of a transaction are that it has an identifier, a date, a cash amount, at most one account from which the money is taken, and at most one account to which the money is transferred.

If you think writing use cases seems easy, you're right. The corollary to this is that if you think you're missing something and that writing use cases should be harder than this chapter makes it appear, you're making the task harder than it needs to be. If writing use cases required more than common sense, the practice would not be successful because you would never get a room full of people to agree on the outcome.

Architect's Exercise: ProjectTrak

ProjectTrak is an open source project management tool. Although some companies have budgets that allow them to buy commercial project management software for their architects and project managers, many do not. Because project management software is an essential tool for any large-scale development, there is a value in having an open source (free) alternative. The ProjectTrak project is hosted at http://projecttrak.sourceforge.net/. A

Figure 2.1: ProjectTrak Use-Case Examples

The system will allow users to define, view, and save project information.

▲ Projects have a name, any number of project tasks, and any number of project resource personnel. In addition, it will track the date the project was created and last updated.

The system will allow users to define, edit, and display project tasks.

▲ A project task has a name, an estimate (in hours), percent complete, one assigned personnel resource, any number of dependent tasks, and a priority (high/medium/low).

The system will compute work schedule information about project tasks.

▲ A projected start date and end date will be computed for each task. This date range will be consistent with the number of working hours available for the assigned resource. This projected date range will not conflict with the range generated for other tasks assigned to this resource.

▲ The order that tasks are completed will be consistent with their priority assignment.

▲ The order that tasks are completed will be consistent with the dependent tasks listed.

subset of the project's use-case analysis is presented in figure 2.1. A complete set of use cases is distributed with the project's source.

Prototyping

At this stage, the development team usually has enough information to choose a user interface technology (which typically involves HTML because most applications are Web-compliant these days). A user interface enables the prototype to become a part of the real application and guards against accidentally promising delivery of something that isn't technically possible.

Consider involving the layout designer in producing the prototype. In fact, the layout designer should almost facilitate this particular exercise instead of the technical architect. I find that technicians usually don't make the most aesthetically pleasing user interface screens. I know I don't.

Remember that prototypes are not functional by definition. None of the prototype screens should have dynamic data. If you are responsible for developing the prototype and are using HTML, I highly recommend Castro (2002).

Further Reading

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

Castro, Elizabeth. 2002. *HTML for the World Wide Web with XHTML and CSS: Visual QuickStart Guide*, 5th ed. Berkeley, CA: Peachpit Press.

Cockburn, Alistair. 2001. *Writing Effective Use Cases*. Boston: Addison-Wesley.

Fowler, Martin, and Kendall Scott. 1997. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley.

3

Scope Definition and Estimation

In most organizations, the project manager works with the business side and management to establish project scope and to estimate time and resource requirements. And frequently, the project manager relies on the technical architect for assistance in these tasks. The scenario is no different for J2EE applications. This chapter is written for architects responsible for helping to define and estimate delivery for management; readers not involved in these tasks can safely skip the chapter.

Defining Scope

Objectively define project scope in terms of use cases, and obtain the agreement of the business side. Changing scope during a project wreaks havoc with project timelines and lowers the morale of the development team. When the business side makes additional requests after development is under way, acknowledge them, record them in a use case, and schedule them for a future release. Often, making rough estimates for each use case provides information the business side will find useful in deciding scope.

Get agreement on the use cases from the project sponsor. As use cases are written in business terms, they can be used as a "contract" with the business side and management as to what will be delivered when. Work with the business side to choose which use cases will be implemented in the current

project. Anything else will be deferred. Even if scope is agreed on verbally, put it in writing and e-mail it to everyone remotely connected to the project. Make sure to keep a copy.

Diligently enforce project scope once it's determined. The most important thing the project manager can do once the scope of a project is determined is to enforce it. The technical architect has the responsibility of alerting the project manager to scope changes. It's much harder to hit a moving target. Although I generally prefer to schedule all enhancements for future releases, the architect usually doesn't get to decide scheduling issues. I estimate the request in use-case form and provide a preliminary estimate. Usually, the project manager can use this information to steer the business side toward scheduling the request for a future release.

Basics of Estimating

Many technical people consider estimating difficult, and indeed, at this stage, it is definitely as much art as it is science. This chapter presents a method of determining a gross estimate based on information you should have from the use-case analysis and the prototypes. I make no claims that my way of estimating is the *only* way. If you have an approach you're comfortable with, stay with it.

Estimates formed at the beginning of a project should be revisited periodically and refined after more detailed planning and designing are done. After defining the external interfaces and completing object and data modeling, you will be in a better position to make more accurate estimates. Estimates you make now should be detailed enough that you can use them in estimating any combination of use cases. Ideally, the business side looks at cost/benefit comparisons when deciding on scope.

Estimate in terms of the slowest resource on the team. We all know that some people take less time on development tasks than do others. I would rather deliver a project early than have to explain why it's going to be late.

Estimates should be proportionally balanced. I have found that a typical development team spends about one-third of its resource budget planning and designing, one-third coding and unit testing, and one-third supporting system- and user-testing activities. Keep in mind that some portion of the planning and design budget was spent performing the use-

case analysis described in the previous chapter (I usually assume that planning and design is 50 percent complete at this point). These ratios apply to the entire life of the project.

Consider the time needed to set up the development, testing, and production environments. Most companies provide established environments at the enterprise level. For example, many companies have a central administration team to establish environments for development, testing, and production of new applications. If your company has such a team, time allocated to environment setup will be low and not usually material for preliminary estimates. If this isn't the case at your company, you should add an estimate for setting up environments.

Developers are more successful at estimating coding and unit-testing tasks than anything else. If you can get a reasonable coding and unit-testing estimate, you can extrapolate the rest using the ratios mentioned previously and get a ballpark estimate. Note that to extrapolate a total estimate based on those ratios, you just multiply the coding and unit-testing estimate by 2.5, assuming that planning and design is 50 percent complete at the time you estimate.

An Algorithm for Estimating

It should be noted that technical architects are responsible for estimating hours only. Project managers should be equipped to account for absences due to responsibilities to other projects and vacation.

Step 1: Determine the number of screens, interfaces, database tables, and conversions for each use case. To derive a coding and unit-testing estimate, gather estimates of the following for the group of use cases being considered for the project:

- ▲ Screens in the user interface (two man-weeks each)
- ▲ External application interfaces (four man-weeks each)
- ▲ Database tables (two man-weeks each)
- ▲ Tables or files conversioned (two man-weeks each)

Not all these items will exist for each use case. The basic estimates noted in parentheses in the preceding list are applicable if the development team hasn't been formed; if there is an existing team, more accurate estimates for each item may be possible. Estimates at this point will not be exact. Estimating

based on the number of objects is more accurate, but that number is not available before object modeling exercises are done.

Step 2: Estimate coding and unit-testing time for each use case. Based on the information gathered in step 1, it's simple mathematics to get a base estimate for a combination of use cases. The *base estimate* is the length of time it takes for one developer to complete coding and unit-testing tasks. It may seem a bit strange to estimate coding and unit-testing time before design is complete. Few developers seem to be bothered by this ambiguity.

Many developers confuse unit testing with system testing. Unit testing is strictly at a class level. How well a class functions when called from other classes within the application is system testing, not unit testing.

For example, for a set of use cases that involves four screens, two external interfaces, five database tables, no data conversions, and two environment setups, the base estimate is:

$$(4 \times 2) + (2 \times 4) + (5 \times 2) + (0 \times 2) = 26 \text{ man-weeks, or } 1{,}040 \text{ hours}$$

Step 3: Multiply the base estimate from step 2 by 2.5 to account for analysis and testing activities for each use case. If coding and unit testing are about one-third of the overall cost of each use case, the total cost should be about three times the base estimate. Because the analysis is about 50% complete at this stage, estimate the total cost to be about 2.5 times the base estimate. Continuing the previous example, the total hours left for the project would be $1{,}040 \times 2.5 = 2{,}600$.

Step 4: Inflate the estimate by 20% for each additional developer on the project. The base estimate assumes that the project has just one developer. Each developer added to a project adds communication and coordination time (Brooks 1975). Although necessary, time spent communicating and coordinating is time not spent developing. Therefore, inflate the base estimate by 20% (i.e., multiply by 1.20) for each developer added. For example, with a base estimate of 2,600 hours and five developers expected, estimate spending $1{,}600 \times (1.20)^4 = 3{,}318$ hours to code and unit test. Incidentally, it's a good idea to round that number to 3,500 to avoid creating the mistaken impression that this is an exact estimate.

Assuming the project's five developers are dedicated full time (32 hours per week, allowing for bureaucratic distractions, etc.), the development team could work a total of 160 hours per week. This means that project delivery

would be approximately five to six months out. Specifically state the estimate in months or quarters at this point to avoid creating the impression that this is an exact estimate.

Step 5: Review your estimates with the development team. If developers don't have input, they won't feel bound by the estimates. It's important to remind developers that these estimates will be reassessed after design is complete.

More manpower rarely saves a late project's timeline. The communication/coordination penalty, as previously described, is the primary reason that adding people to a late project only makes it later. Architects are often asked for ballpark estimates before all the analysis is done. Even if it won't be literally correct, you can ballpark an estimate by filling out with assumptions and documenting them with the estimate.

Architect's Exercise: ProjectTrak

Based on the requirements documented in the use cases we identified in chapter 2, we would expect screens or pages for the following topics:

- ▲ Project definition view/edit
- ▲ Task definition view/edit
- ▲ Resource definition view/edit
- ▲ Resource calendar definition view/edit
- ▲ Skill set view/edit
- ▲ Project work schedule view and calculation
- ▲ Progress report view
- ▲ Gantt chart view

Some of these pages will be more complicated than others, but at this stage, we don't need to worry about that. Let's assume that it will average out. Working with an estimate of two man-weeks for each page, let's add sixteen man-weeks to the estimate.

There are no external interfaces or data conversions for this product, so we don't need to add time to the estimate for these activities.

Based on the requirements, at a minimum, we will have database tables for the following constructs:

- ▲ Project
- ▲ Task
- ▲ Task resource assignment
- ▲ Resource
- ▲ Resource work schedule
- ▲ Skill set
- ▲ Resource skill set

Working with an estimate of two man-weeks per table, let's add fourteen man-weeks to the estimate.

The base estimate is then approximately 30 man-weeks, or 1,200 hours of coding and unit-testing time. Because this is only coding and testing time, we estimate the total number of hours left for the project at 1,200 × 2.5 = 3,000 hours.

Assuming that three developers will be assigned to the project and allowing time for communication and management, we estimate $3{,}000 \times (1.20)^3 = 5{,}184$ hours. As discussed, we'll want to round that to 5,000 hours so as not to create the impression that this is an exact estimate.

Further Reading

Brooks, Frederick P., Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley.

DeMarco, Tom, and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams*, 2nd ed. New York: Dorset House, 1999.

4

Designing External Application Interfaces

It is common for a J2EE application to communicate with external applications. For example, a purchasing application may notify an accounting application of all purchases, or an inventory management application may notify an accounting application of all inventory receipts and customer shipments. A technical architect is responsible for the design of application interfaces as well as the application itself. This chapter describes how to define external application interfaces in enough detail that both the J2EE and external applications will have the information needed to perform design and implementation tasks.

If your J2EE application programmatically initiates processing in external applications, you should identify those applications as actors in the use-case analysis, as discussed in chapter 2. All specifics about the information transferred between the J2EE application and the external applications should be the subject of one or more use cases. For example, the fact that a purchasing system notifies the accounting system about all orders placed should be the subject of a use case.

Use cases for an external application interface should identify the events that trigger use of the interface as well as the information passed for each event. For example, consider the following use case:

The inventory system will notify the accounting system about all inventory receipts.

- ▲ Notification will occur immediately after the receipt is recorded.
- ▲ Notification will include the vendor ID and timestamp for each receipt as well as the UPC and quantity for each item in the shipment.
- ▲ Confirmation of the notification is required from the accounting system.

Formally define and document the external interfaces so that developers from both applications have a basis for object-modeling activities (described in chapter 6). The technical architects for both applications need a basis for their modeling and implementation activities. Further, the project manager needs a contract between your development group and the external system developers that describes everyone's responsibilities.

The following aspects of external interfaces should be discussed with the external teams and agreed on among the teams:

- ▲ Communication method
- ▲ Data content structure
- ▲ Triggering events for content exchange
- ▲ Error-handling procedures and responsibilities

The technical architect should facilitate interface design discussions. Part of the role of facilitator is keeping the discussions limited to the topics just listed. The internal design of either application is largely irrelevant to this discussion. The platform used by the external application is relevant because it could affect the ability of the J2EE application to use some communication methods. For example, if the external application is not written in Java, the J2EE application cannot use any form of communication that uses RMI, such as enterprise beans.

Selecting a Communication Method

Communication with another application is either synchronous or asynchronous. With *synchronous communication*, the transmission occurs immediately and an error is generated if the external application is down. Examples of synchronous communication technologies include EJBs, RMI, Web services, CORBA, and HTTP. And I have even seen some projects use HTTP

programmatically as a method for communication. With *asynchronous communication*, application processing continues while the communication is being completed. The application your communication is directed at may not receive your message immediately. Examples of technologies that can be used asynchronously include messaging/JMS and Web services.

Whether you choose synchronous or asynchronous communication depends on the business requirements of the application. Use cases requiring external interfaces should describe the business process in enough detail that you can make this choice. If the external application requires immediate action, your best choice is synchronous communication. On the other hand, if delayed processing is acceptable from a business perspective, your best choice is asynchronous communication. The following sections provide more detail on each communication method.

Asynchronous Communication

Asynchronous communication is typically implemented via messaging technologies, known in the Java world as JMS. You can also use asynchronous communication for broadcasts, commonly known as publish/subscribe capability. That is, you can send a message to any number of applications that care to listen. The message content is usually informational rather than some type of processing instruction.

Depending on your messaging vendor, messaging can be platform independent. Most of my clients use IBM's MQ/Series. MQ has client interfaces for most platforms and would be a viable choice for communicating between a Windows .Net application and a J2EE application, for example.

Messaging technologies have very loose coupling. Applications can generally make any technology change the way they want to without affecting messaging interfaces, as long as they continue to support the message transmission protocol and format. For example, you could convert one of the applications from COBOL to Java without requiring any changes to other message senders or receivers.

Synchronous Communication

Although you can implement synchronous communication with messaging, it is more common to implement it using Web services: RMI/EJB, CORBA, or HTTP. If you use messaging for synchronous communication, it is generally point-to-point, not any type of broadcast. Synchronous communication is generally configured to generate an error if the external application is not available and structured to require a response. You can think of

synchronous communication as an external call. Web services are just different ways to structure that call.

Using messaging in a synchronous fashion confuses some people, but it is common in the healthcare industry. I've also seen it used in utility companies as well. Consider the following example of how a purchasing system uses synchronous messaging to obtain available credit information from an accounting application:

1. The purchasing system creates an XML document containing instructions to return the amount of credit available to a specific customer.
2. The purchasing system sends this XML document as a message to a specific queue monitored by the accounting application.
3. The purchasing system waits for a response from the accounting application.
4. The accounting application receives the message, parses the XML text, and obtains the needed credit information.
5. The accounting application creates an XML document with the credit information and responds to the message received, incorporating the XML text in the response.
6. The purchasing system receives the response, parses the XML document created by the accounting application, and continues processing.

Increasingly, Web services are being used for synchronous communication. The primary advantages of Web services are (1) they are platform independent to the point that the external application platform is not relevant, and (2) they are more loosely coupled than CORBA or RMI/EJB calls. Web services are a good choice for synchronously communicating with non-Java applications (such as .Net applications).

Using HTTP to communicate with another application has similar advantages and disadvantages to using Web services because they both use HTTP. HTTP communication does require that you adopt some type of application protocol because it doesn't natively use SOAP.

To understand the concept of using HTTP as a communication method, consider the following example of a purchasing application using HTTP to request information from an inventory management system:

1. The purchasing application issues an HTTP request (using java.net.URL) to the inventory application requesting current inventory levels for an item (e.g., laundry detergent).
2. A servlet in the inventory management system receives the request and initiates processing.
3. The inventory management system examines its database and determines the quantity of the requested item available at all warehouses.
4. The inventory management system constructs an XML document containing the quantity information.
5. The servlet in the inventory management system returns the XML document to the caller in the same way it would return HTML to a browser.
6. The purchasing application receives the XML document, parses it, and continues processing using the inventory information received.

In this example, the XML format used must have the same design as XML used with messaging technologies.

CORBA allows external calls to applications written on any platform that supports CORBA. In this sense, CORBA is more platform independent than is RMI/EJB but less independent than Web services or HTTP. CORBA is also slightly more mature than Web services or RMI/EJB.

Both RMI services and J2EE enterprise beans are restricted to Java applications. This tight coupling between applications can create deployment difficulties because both applications need to use compatible versions of common classes to avoid marshalling errors.

All types of synchronous communication require an error-processing strategy if the external application is unavailable or a transaction cannot be processed properly. Your options are either to produce an error or to develop a mechanism within your own application to retry later. I prefer the latter when possible because, although this does introduce complexity into the application, the administrative overhead dealing with outages is too high otherwise.

Comparing the Two Methods

Let's look at some examples of each method. If you send a synchronous message or call another application, giving it an instruction to do something for a customer, your application waits for that transmission to occur before

continuing processing. Consequently, your application will know when there is a communication failure. If a response is expected, your application will know when processing errors occur with the request. Unfortunately, because your application is waiting for the communication and possibly a response, there is a portion of application performance you don't control.

If you sent the same message asynchronously, your application would not wait for the communication. Because the communication physically happens *after* an application initiates it, it often appears faster than synchronous communication. And asynchronous communication is more fault tolerant. If the target application is temporarily down, the message will be delivered automatically when that application comes back up, with no manual intervention. The disadvantage is that asynchronous communication cannot detect processing errors and does not know when your transaction will be physically processed.

Table 4.1 summarizes the features of each method.

Table 4.1: Features of Synchronous and Asynchronous Communication Methods

Feature	EJB	Web Services	Messaging/ JMS	RMI	HTTP	CORBA
Caller platform requirements	Java-compliant only	Any	Any	Java-compliant only	Any	Any
Communication method supported	Synch. only	Both	Both	Synch. only	Synch. only	Synch. only
Coupling	Tight	Loose	Loose	Tight	Loose	Loose
Transaction support	Local and JTA	Local and JTA	Local	Local	Local	Local
Requires J2EE container?	Yes	No	No	No	No	No
Support clustering for scalability and availability?	Yes	Yes	Yes	No	Yes	Yes

Common Mistakes

Using databases and file systems as "message brokers." This error-prone strategy consists of writing a row in a database table (or file in a file system) with the contents of a message. The second part to the strategy is writing an agent that "polls" the database (or file system) for new messages to arrive, reads and processes the message content, and then deletes the row or file.

In essence, this strategy is akin to writing your own messaging system. It is typically a frequent source of bugs. Why reinvent the wheel? Choose one of the existing forms of communication and concentrate on business logic, not low-level communication programming.

Using an asynchronous communication method, such as messaging, when a response is required. With this strategy, your application sends an asynchronous message to another application and waits for a return message from that application.

Using asynchronous communication when responses are required puts you in the position of programming an algorithm to wait for a response. How long do you wait? If you wait too long, you could be holding up a user. If you don't wait long enough, you could mistakenly report an error. This is akin to two blindfolded people trying to find each other in a vacuum.

As asynchronous messaging requires messaging software, using asynchronous communication when a response is required adds components and complexity to the application. Although messaging technologies are robust and stable, synchronous communication methods are less vulnerable to unplanned outages.

Determining a Data Structure

Document all data passed to an external application. The communication structure or format, along with the conditions that dictate transmission, is part of your contract with the external system developers. The data structure should be documented and agreed to in writing.

If the communication method chosen is Web services, RMI, CORBA, or enterprise beans, documenting the data structure is an object-modeling exercise. You'll need to fully define the services or beans along any method, as well as the objects used in their arguments. Object-modeling concepts are more fully covered in chapter 6.

The parts you need to identify and document for the external interface are the name and type of the service, the name and return type (typically a

value object) of the method, and any required arguments. Since the return type and arguments might be value objects (VOs), the fields on those VOs will have to be documented as well. Sometimes VOs are referred to as *data transfer objects*.

For example, consider a Web service, CustomerService, with a method called `getCustomerInfo()` passing a userId string and returning an object of type `CustomerVO`. For the external interface, all parties need to know legal argument rules, what the fields in `CustomerVO` will contain, and what exceptions, if any, might be thrown. Figure 4.1 shows an example.

If you choose messaging as the communication method, you need to fully define the format of the message transmission. By far the most popular format employed these days is XML.

Figure 4.1: Example Interface Specification

Service Name: CustomerService
Service Type: WebService
Client class: com.jmu.client.CustomerServiceClient
Client jar: JmuClient.jar

Customer Information Retrieval

Method Name: getCustomerInfo

Arguments: CustomerID—(String)
- ▲ CustomerID cannot be null
- ▲ CustomerID cannot be blank

Returns: com.jmu.vo.CustomerVO
- ▲ firstName (String)
- ▲ lastName (String)
- ▲ id (String)
- ▲ streetAddress (String)
- ▲ city (String)
- ▲ state (String)—two letter abbreviation, capitalized.
- ▲ telephone (String)

Exceptions:
- ▲ java.lang.IllegalArgumentException if customer ID is null or blank
- ▲ com.jmu.common.CustomerNotFound if no customer exists
- ▲ java.rmi.RemoteException if technical issue with the call

XML is a common protocol for interapplication communication and message formats. XML is preferred in the Java world because many open source tools are available.

Use simple XML formats for interfaces to legacy platforms. XML can be a headache for developers using legacy platforms if your company won't buy an XML parser. Open source parsers and other tools are not available for COBOL or PL/I, at the time of this writing. This would be a valuable open source project.

When the external application is a legacy system and the client doesn't buy XML tools to assist, custom formats are common out of practical necessity. Custom formats can have a keyword-type organization (like XML) or some type of positional organization. COBOL lends itself to fixed-length strings using a positional format.

You need to develop a DTD or schema, or otherwise document the XML document formats and tags with allowed values. Leaving this communication verbal is a recipe for disappointment and countless project delays. Chapter 7 includes material to help you design XML documents.

You don't necessarily have to validate XML documents for external interfaces. Developing a DTD to describe a document format does not mean that you have to "validate" the document when it's parsed. Validation works well for documents that were directly written by people, and thus more likely to contain errors. Application interfaces are mechanical. Aside from initial development, the probability of receiving a malformed XML document that was programmatically generated from another application is low. Given this, the benefits of validation don't usually outweigh its performance costs.

Avoid sending serialized Java objects as message content. This effectively negates the benefits of loose coupling by making both applications dependent on the same classes. If a change is made to one of the objects referenced by the serialized class, the change would have to be deployed to both applications simultaneously. Further, serialization problems can occur if one of the applications upgrades its Java Virtual Machine (JVM). Serialized objects cannot be visually inspected to determine content; they must be programmatically processed. They also limit communication to Java applications.

Error-Handling Requirements

Error-handling requirements need to be discussed, documented, and agreed to just as thoroughly as data formats do. Bad error handling will lead to high maintenance costs.

Error Notification Procedures

All external application interfaces should have error notification procedures. Some organizations provide centralized operations personnel responsible for notifying application support personnel of errors. This is often achieved through some type of console message. It's not uncommon for the console to be monitored 24x7x365. It's normal for organizations with this capability to provide some way to programmatically generate a console message that someone takes action on.

Use mechanical error notification. In companies that haven't established enterprise-wide error notification procedures, I typically include some type of e-mail notification for severe system errors. Merely writing errors to a log assumes that someone will look at them. Most messages written to the log often go unnoticed. As most alphanumeric pagers accept messages via e-mail, it's easy to include pager notification for severe errors. This should be used with caution for obvious reasons.

Don't be afraid to be verbose when logging for errors. My philosophy is that the error log should contain as much information as possible. While some of it might not be useful for a particular problem, it's better to have too much than too little when it comes to information. If logging is successful, a large percentage of errors should be diagnosable just from the error message without further need to reproduce the problem.

At some companies, commercial tool support is available to help with logging. BMC Patrol, EcoTools, and Tivoli are commercial network management toolsets that are often used to monitor the enterprise. OpenNMS (http://www.opennms.org/) is an open source alternative for network management.

Retry Procedures

Once initial development is complete, most errors with interfaces have environmental causes. Examples include someone recycling the database or messaging software, someone tripping over a network cord, and a server with a full file system. In most cases, the error is eventually fixed and the application interface resumes proper function.

However, with most applications, some type of recovery or retransmission procedure must be performed for the transmissions that failed. For example, messages recording customer purchases from the ordering application must be resent to keep accounting records accurate.

Make retry procedures mechanical. For transmissions that a user isn't physically waiting for, I often put a scheme for automatic retry logic in place. Environmental problems occur often enough in most companies to warrant including selective retry logic to recover from outages more quickly and with less effort. The objective of the retry mechanism is to automatically recover from a temporary outage without manual intervention by an application administrator.

The potential for complexity lies in discriminating between temporary outages and errors that won't magically go away over time. It's possible to take a shotgun approach and assume that all errors occur as a result of a temporary outage and initiate retry logic. You should use this approach with care.

All mechanical retry logic should have limits. It's important not to retry forever, essentially creating an infinite loop. Choose a sensible retry interval and number of retry attempts. It's also wise to make these limits configurable so they can be easily adjusted. Retry attempts should also be logged or possibly follow your mechanical notification procedure. Upon receiving notification of the retries, an application administrator might realize that corrective action should be taken.

External Interface Guidelines

Many J2EE applications communicate with external systems, some of which may not be Java applications. As a technical architect, you'll probably have an opportunity to facilitate design and implementation of external interfaces. Over the years, I've adopted several techniques for creating successful external interfaces.

Record every request or transmission from or to an external application. This should be the first task in any external interface. Be sure to log this information so that you know the time (and node if you're in a clustered environment). If there's a problem with work you initiated in an external application, you'll want to be able to tell those developers exactly what calls they made and when. If there's a problem processing a call from an external application, you'll want enough information to replicate it and fix any problems quickly.

This documentation practice helps prevent you or any member of the development team from being blamed for the mistakes of others. Without facts, anyone can make accusations, and in too many corporations, you're guilty until proven innocent. Having a transmission log will make it easier to determine which application has a problem.

Create a way for an application administrator to resubmit a work request from an external application. When your application experiences an environmental problem and you're able to fix it, the application administrator should be able to resubmit the work that failed. This limits the impact on end users and limits the number of people (and thus the time) needed to fix a problem. Sometimes, resubmission isn't possible or applicable. But it can save you time for activities that are more fun than maintenance.

As an example, one of the applications I supported provided scheduled reporting capabilities. An application administrator could resubmit any report that wasn't being delivered to a browser by an administrative Web-based utility.

Collect the identity of the transmitter. Part of every transmission should be information that indicates the identity of the caller. This is meant as a way to reduce your maintenance time, not as a security mechanism. In the case of an inappropriate call, you want to be able to quickly notify the application making it. If you don't know where it's coming from, finding it and getting it fixed takes longer.

Develop a mechanical way to "push" errors to the transmitter. If there is a problem processing a request made by an external application, administrators from that application need to be notified. If the interface has a tight coupling (e.g., session bean, RMI service), all you have to do is throw an exception. Whatever mechanism the external application uses for error processing will be able to handle the error.

If the interface has a loose coupling (e.g., uses messaging technologies), you need to construct a way to mechanically notify administrators from the external application of the error. Most often, I've seen e-mail used as an effective error notification channel for these cases. As most alphanumeric pages accept e-mails, paging is a possibility. However, I would thoroughly test your error processing before hooking it up to a pager.

Architect's Exercise

I assisted in the development of a customized reporting system that accepted information via JMS from external applications. In this case, the two types of information being received were batch-processing requests and updates to meta-data needed to produce reports.

In all cases, the format of the messages received was XML. We had different document formats for the batch-processing requests and the report template meta-data updates. Immediately after receiving a message, we would go through the following steps:

1. Parse the XML document.
2. If there was a parse error, log the error and the contents of the message as well as notify an application administrator by e-mail.
3. If the document was well formed, interrogate the document type.
4. If the document was one of the two supported types, record it's receipt, including the XML text, the document type, and the date and time of receipt, in a file on disk (using a database or any other storage media).
5. Process the request normally, mailing any errors, along with the XML text, to an application administrator mail group. The application sending us the XML messages had representatives in the mail group.
6. An application administrator could resubmit both types of requests via a secured Web page if necessary.

Section 2

Designing J2EE Applications

The technical architect typically leads and facilitates all application design activities. In this section, you will learn how to:

- Document application designs with object models so other development staff can easily understand the design.
- Understand software layering concepts and how to use them to organize J2EE applications into smaller and more manageable sections.
- Apply common design patterns at each software layer.
- Document application storage requirements using data-modeling techniques.
- Design XML document formats.
- Understand common network architectures for J2EE applications and how the network architecture assists in providing security, scalability, and high-availability features.
- Refine project estimates after the design phase.

5

A Layered Approach to J2EE Design

Object modeling is a key skill for any technical architect with any object-oriented technology, such as J2EE. Object models are the most popular mechanism for documenting J2EE application designs. The technical architect is typically responsible for facilitating creation of an object model. This chapter presents and explains a general approach to designing J2EE applications. This discussion is necessary background to the next chapter, which offers tips and techniques for transforming the use-case analysis described in previous chapters into a concrete design.

J2EE application design is a vast topic. Entire books have been written about design techniques, design patterns, and object modeling with UML. And like most large and complicated methodologies, UML is only partially applied in business applications. Thus, although I don't want to discourage learning, I do take a streamline approach to the topic in this chapter. For instance, of the hundreds of design patterns that have been identified and cataloged, this chapter focuses on the handful of patterns most commonly used in business applications today.

Overview of the Layering Concept

A common framework for J2EE applications is *software layering*, in which each layer provides functionality for one section of the system. The layers are organized to provide support and base functionality for other layers. For

example, the data access layer provides a set of services to read and write application data. An inventory management application needs services that can read information about specific items and warehouses.

Layering is not a new concept; operating systems and network protocols have been using it for years. For instance, anyone who's worked with networks is familiar with telnet, FTP, and Internet browsers. All these services depend on a TCP/IP networking layer. As long as the interface to TCP/IP services stays constant, you can make network software advances within the TCP framework without affecting telnet, FTP, or your Web-browsing capability. Typically, the TCP/IP layer requires the services of a device layer, which understands how to communicate with an Ethernet card.

Applications can make strategic use of the same concept. For example, it's common to make data access a separate portion of an application so that data sources (e.g., Sybase or Oracle) can be changed relatively easily without affecting the other layers. The way the application physically reads and writes data may change without affecting application processing or business logic.

To continue the inventory management example, suppose the data access layer had a method to look up the information for an inventory item given a UPC code. Suppose that other parts of the application use this method when information about items is needed (e.g., a customer viewing that item on a Web page or an inventory analyst ordering stock for that item). As long as the methods for access item information remain the same, you should be able to reorganize how you can store information about items and warehouses without affecting the rest of the application.

In essence, a layered approach mitigates the risk of technical evolution. If you use this concept to separate your deployment mechanics (e.g., servlets, enterprise beans), you can add new deployments without changing your business logic or data access layer. For example, Web services have become popular only within the last two years. If your application effectively separates your business logic and data access logic from the rest of your application, you can freely add a Web services deployment without having to change the entire application.

Table 5.1 lists common software layers used in J2EE applications. You can think of these layers as "roles." For instance, a customer object may have a role in the data access layer, the business logic layer, and the deployment layer.

Table 5.1: Roles of Software Layers in J2EE Applications

Layer	Role
Data access object layer	Manages reading, writing, updating, and deleted stored data. Commonly contains JDBC code, but could also be used for XML document and file storage manipulation.
Business logic layer	Manages business processing rules and logic.
Value objects layer	Lightweight structures for related business information. These are sometimes referred to as *data transfer objects*.
Deployment layer	Publishes business object capabilities.
Presentation layer	Controls display to the end user.
Architectural component layer	Generic application utilities. Often, these objects are good candidates for enterprise-wide use.

Figure 5.1 illustrates how the individual software layers interrelate.

When this boils down to code, I usually implement the layers as separate packages. Here's an example package structure:

`com.jmu.app.dao`	Data access object layer
`com.jmu.app.bus`	Business logic layer
`com.jmu.app.vo`	Value objects layer
`com.jmu.app.client`	Presentation layer
`com.jmu.app.util`	Architectural component layer
`com.jmu.app.deploy`	Deployment layer

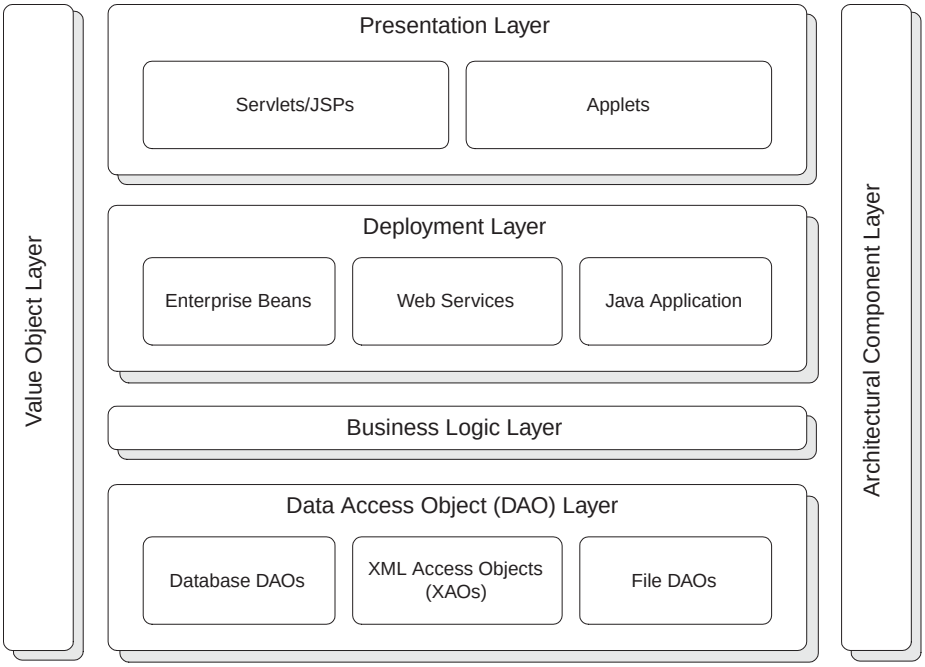
I use the abbreviation `jmu` for "just made up." Also, you'll want to replace the `app` abbreviation with a meaningful application name.

One question I commonly hear is, why not call the data access object layer directly from the presentation layer? Although calling the data access object layer directly can save some code by eliminating a couple of layers, it means placing any business logic either in the data access object layer or the presentation layer, which makes those layers more complex. Software layering works on the premise that it's easier to solve multiple small problems than fewer large ones. Experience has taught me two lessons:

- Every time I try to eliminate software layers, I end up having to come back and restructure the application later.
- There is tremendous value in consistency.

There is value in consistency for maintenance purposes. For instance, if some JSPs call data access objects directly while others work through business

Figure 5.1: Software Layers for J2EE Applications



objects and others use deployment wrappers, to make a change to code you're unfamiliar with you have to do an audit of the whole action sequence. This defeats the purpose of object orientation.

The remainder of the chapter discusses each layer in depth and common design patterns used for each.

Data Access Object Layer

Data access objects (DAOs) manage access to persistent storage of some type. Usually, the storage used is a relational database, but DAOs can manage files, XML documents, and other types of persistent storage as well.

The primary reasons to separate data access from the rest of the application is that it's easier to switch data sources and share DAOs between applications. Medium- to large-sized businesses in particular are likely to have multiple applications using the same data access logic.

A couple of patterns for data access objects are most common. The simplest pattern has each persistent object represented as a DAO. I call this the simplified data access pattern. The more complex, but more flexible, pattern

in common use is a factory-based pattern. In fact, it's called the data access object pattern. I'll define each pattern later in the section.

For convenience, I separate DAO objects in the package hierarchy (e.g., `com.acme.appname.data` or `com.acme.appname.dao`). I only mention this because some modeling tools (e.g., Rational Rose) encourage you to decide your package structure at modeling time. Some developers also add a DAO suffix to data access object names; for example, a customer DAO might be named `CustomerDAO`.

Choosing a Database Persistence Method

The question of which persistence method is best is the subject of considerable disagreement and debate. Although the J2EE specification provides for entity beans, other popular forms of database persistence exist. The degree to which developers take sides in the debate is akin to a discussion of religion or politics. The debate is not entirely rational. I'll take you through my thoughts on the different options, what I see in the marketplace, and what I prefer to use. However, the modeling concepts in this chapter are applicable to all persistence methods.

When beginning object-modeling activities, you can identify a DAO without choosing the persistence method you'll use at implementation. For instance, the DAO could be a custom-coded JDBC class, an entity bean (EJB) with bean-managed persistence (BMP) or container-managed persistence (CMP), a JDO object, or an object generated by an object-relational (O/R) mapping tool such as TopLink or Hibernate. J2EE applications are compatible with all these persistence methods. However, you should choose a persistence method before completing object-modeling activities, because the method you choose can affect the design.

In making a decision, I first consider what needs to happen at the data access objects layer. I then grade each persistence method according to how well it achieves the goals, using the following rating system:

- ▲ High (best rating): Gets high marks toward achieving the stated goal
- ▲ Medium (middle rating): Moderately achieves the stated goal
- ▲ Low (lowest rating): Doesn't achieve the stated goal very well

Table 5.2 lists the goals and ratings of several data persistence methods. Following the table are explanations of my reasoning in determining the ratings. I consider the first four goals listed in the table to be the most important to the majority of my clients.

Table 5.2: Ratings of Data Persistence Methods

Goal	JDBC	EJB/BMP	EJB/CMP	JDO	O/R Tool
Minimize learning curve	High	Low	Low	Medium	Medium
Minimize code and configuration files written and maintained	Low	Low	Low	Medium	Medium
Maximize ability to tune	High	Medium	Low	Low	Low
Minimize deployment effort	High	Low	Low	Medium	Medium
Maximize code portability	Medium	Medium	High	High	High
Minimize vendor reliance	High	Medium	Medium	Medium	Low
Maximize availability and fail-over	Low	High	High	Low	Low
Manageable via JTA	Yes	Yes	Yes	Yes	Yes

Minimize the learning curve. Because JDBC was the first persistence API for databases, it is the most familiar to most, if not all, Java developers and thus has the lowest learning curve. The learning curve for entity beans with container-managed persistence is widely acknowledged to be large. To use entity beans with bean-managed persistence, developers need to understand both JDBC and entity beans. JDO and most O/R toolsets have learning curves that are higher than JDBC and lower than entity beans.

Minimize code and configuration files written and maintained. People have a tendency to consider the number of lines of code only when evaluating ease of development. I view any configuration file (e.g., an entity bean deployment descriptor) as code with a different syntax. Hence, I don't see entity beans as having less code or simpler code than JDBC. JDO and most O/R toolsets I'm familiar with save some percentage of code in most situations.

Maximize the ability to tune. Because it's the lowest level API and closest to the database, JDBC provides unfettered ability to tune database SQL. Every other choice relies on a product to generate the SQL used. For instance, most object-relational mapping tools generate the SQL executed; it's usually harder to tune without direct control of the SQL used.

Minimize the deployment effort. Deployment hassles negatively impact development and maintenance time. Changes in JDBC code require just a recompile, whereas entity bean changes require a recompile, stub generation,

and redeployment of the bean. JDO implementations and O/R toolsets may require more work than a recompile but usually less than a bean change.

Maximize code portability. Being able to easily port code to different databases is important. Although you can write portable JDBC code, a significant percentage of JDBC code uses some database feature that's proprietary and not portable. Entity beans with BMP contain JDBC code and get the same grade. Entity beans with CMP don't have JDBC code, but they do have mappings and sometimes SQL statements in the deployment descriptors. If the SQL statements use a proprietary database feature, there's a portability issue, but the magnitude of the problem would likely be smaller. The same could be said of most JDO and O/R toolset implementations.

Minimize vendor reliance. You need to reduce your dependence on vendors that provide J2EE container services, JDBC drivers, JDO drivers, and O/R toolsets. This is desirable from a business standpoint should a vendor fail or change its cost structure. You can change vendors for JDBC drivers easily; I've done it many times with multiple database vendors.

Changing container vendors is moderately easy. You do have the possibility of performance-tuning issues, particularly with CMP and JDO vendors. With O/R toolsets, application code directly uses vendor (or vendor-generated) classes. Switching O/R toolsets requires significant development in most cases.

Maximize availability and fail-over. Most of the time, availability and fail-over are provided by the container or database vendor. With the possible exception of entity beans, the persistence method adds absolutely nothing to fail-over. Entity beans, both CMP and BMP, are excepted because the container has more control and can provide better fail-over services. However, fail-over capabilities in this regard are largely dependent on your container vendor.

Manageable via JTA. The Java Transaction API (JTA) is the part of the J2EE specification that provides the two-phase commit functionality. Many developers appear to be under the impression that to use two-phase commit functionality, you need to use entity beans. That's not true. As long as you're able to manage transaction commits and rollbacks via JTA, you can get the two-phased commit functionality.

Although I've seen selected use of entity beans, JDO, and O/R toolsets, most of my clients manage persistence using native JDBC. JDO use seems to

be getting more press and may indeed be the fastest-growing JDBC alternative, but it hasn't pervaded the market yet.

I think about the use of software toolsets in the same way an economist thinks about market efficiency. Financial analysts and economists have a theory that financial markets are efficient. That is, when a new piece of information becomes public, the stock prices of all companies related to that information change accordingly over time. For example, when the Enron and United Airlines bankruptcies became publicly known, the news had profound effects on their stock prices.

When new software paradigms are introduced, if they provide benefits that exceed their costs, over time developers will switch to them. The time frame in which this happens for programming paradigms is much slower than that for financial markets, but the general concept is much the same. The "market" consensus regarding database persistence appears to be favoring native JDBC at the moment. If developers do in fact migrate to the more productive software paradigms over time, the inference would be that native JDBC persistence is the best choice for most applications.

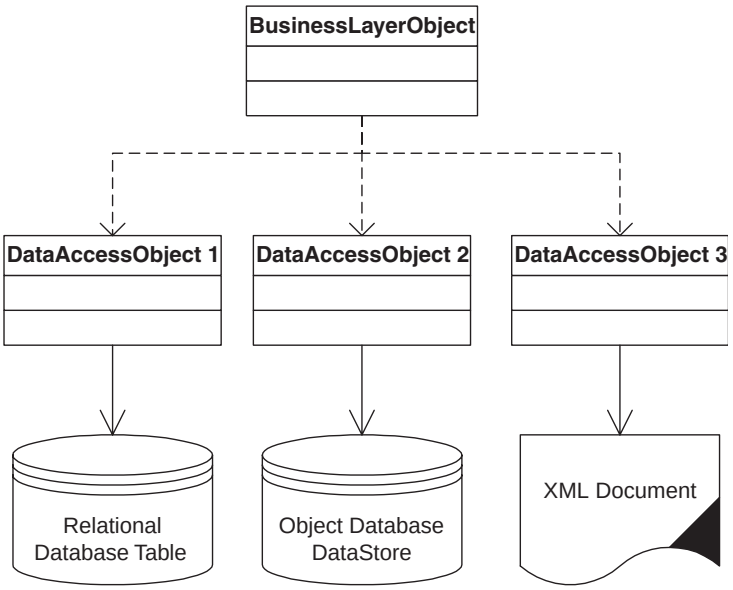
Simplified Data Access Pattern

Of the two patterns for data access objects that are most common, this is the simplest. In this pattern, there is a one-to-one correspondence between the physical storage construct (e.g., relational database table, XML document, or file) and the DAO that manages it. For example, you might have `CUSTOMER_DAO` manage access to a `CUSTOMER` table in a relational database. Although you haven't identified methods yet, you can imagine that this class will have methods to search and return information on one or more customers using search criteria as arguments. It might have methods to insert, update, and delete customers as well.

The advantage of this pattern is that it's simple. Its chief disadvantage is that it's often specific to one data source type. The code involved in manipulating an XML document is quite different from the JDBC and SQL required to use a database table. Switching the data source type would be a major overhaul to most methods in the class.

This pattern is usable no matter what database persistence mechanism you choose. If data access will be managed by an entity bean, that entity bean would in essence be your DAO. The same could be said for a class using native JDBC, JDO, or an O/R toolset. Figure 5.2 illustrates an object model for this pattern.

Figure 5.2: A Simplified Data Access Pattern

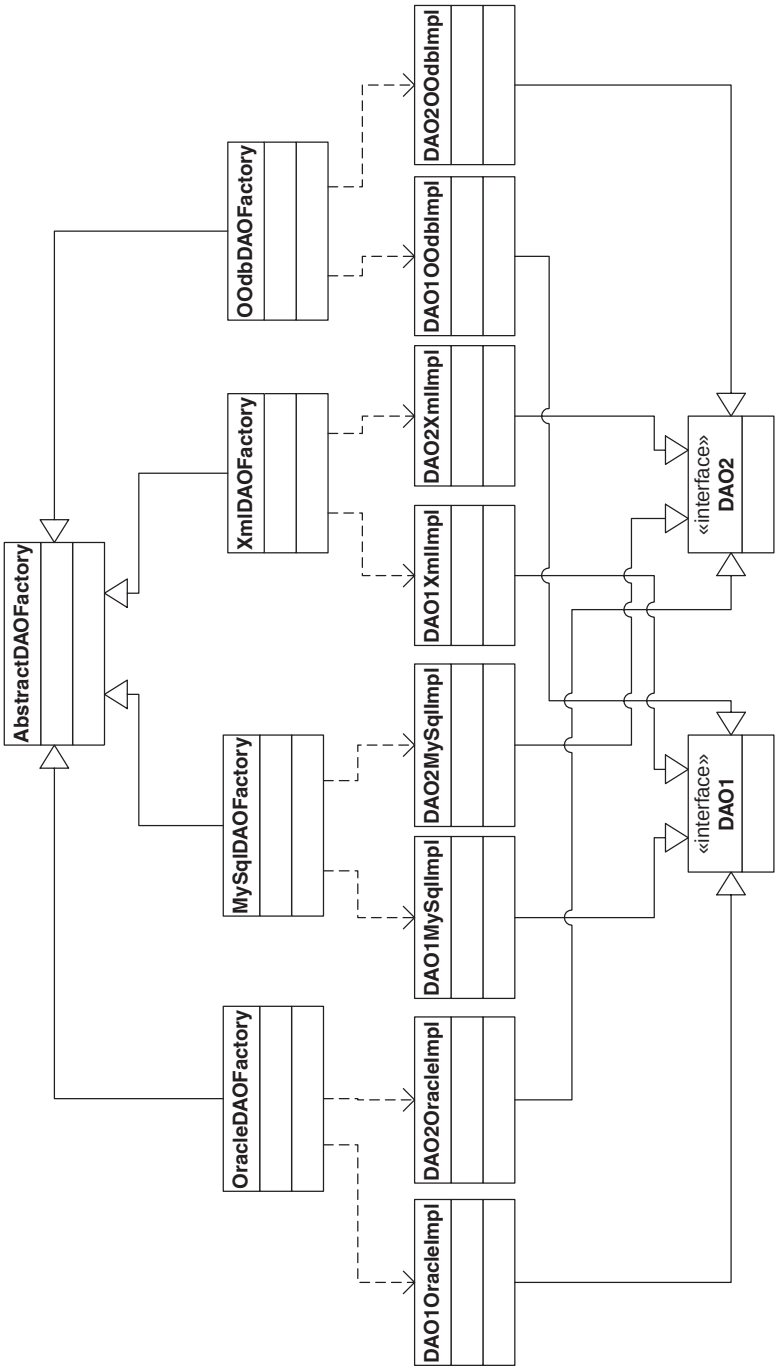


Supporting Multiple Databases

For an application that supports multiple types of databases, the data access object pattern, which is factory based, is quite common. This pattern implements the DAO as an interface. A factory is required to produce objects that implement this interface. In addition, you will have an implementation of this interface for each type of data source. The factory is smart enough to know how to instantiate all implementations. Figure 5.3 illustrates an object model for this pattern.

For example, consider a customer DAO implementation. It would have a `CustomerDAO` interface that would specify a variety of search methods as well as an update, delete, and insert method. It would also have a customer DAO factory (`CustomerDAOFactory`) responsible for providing a DAO for the application to use. It might also have an implementation for all relational databases it supports (e.g., `CustomerDAOOracleImpl`, `CustomerDAOSybaseImpl`, `CustomerDAOMySQLImpl`, etc.). The business object code would use the `CustomerDAO` interface exclusively so it could use any one of the implementations.

Figure 5.3: Data Access Object Pattern



The data access object pattern is overkill if you don't foresee a need to support multiple database vendors. Few business applications need to be able to do this. Software vendors are more likely to need this pattern than business application users. I discuss it at length because it has an unqualified recommendation in many texts.

Value Object Layer

Every application has data items that logically belong and typically are used together. It's programmatically convenient and, with enterprise beans, performance enhancing to treat this logical group of data items as a separate object. This type of object is commonly known as a value object (VO), although some texts refer to it as a data transfer object. If Java had a "structure" construct, as C/C++ and many other languages do, a VO would be a structure.

For example, you could combine various pieces of information for a report template into a VO. Methods needing a report template argument could then accept the `ReportTemplateVO` instead of individual arguments for all components of the structure.

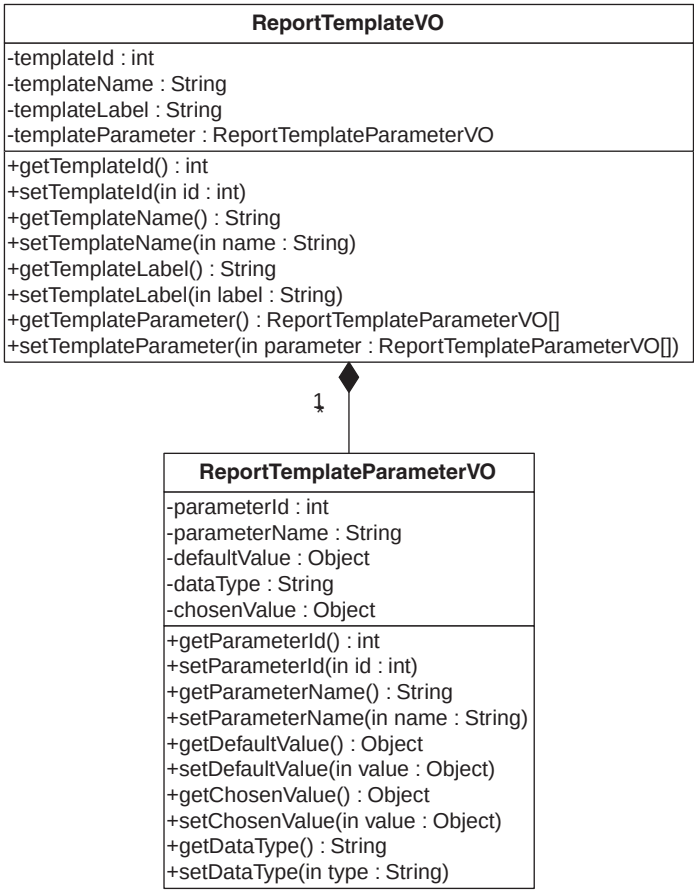
Typically, a VO has accessors and mutators but little else. And usually a VO implements `java.io.Serializable` so it can be transmitted to remote application clients. J2EE containers and RMI services serialize the content of Java classes before transmitting them to a remote machine. A common convention is to give value object names a VO suffix, as in `CustomerVO`.

Common Patterns

The value object originates from a formally defined pattern. In some texts, this pattern is called the value object pattern (Alur, Crupi, and Malks, 2001). The VO pattern enhances EJB performance but is useful in communication among all layers of the application.

You can combine the VO pattern with the composite pattern, which is used when something contains other things. For instance, a report template often contains multiple parameters. Using the composite pattern, the `ReportTemplateVO` contains an array of `ReportTemplateParameterVO` objects, as illustrated in figure 5.4.

Figure 5.4: Composite Pattern with Value Object Context

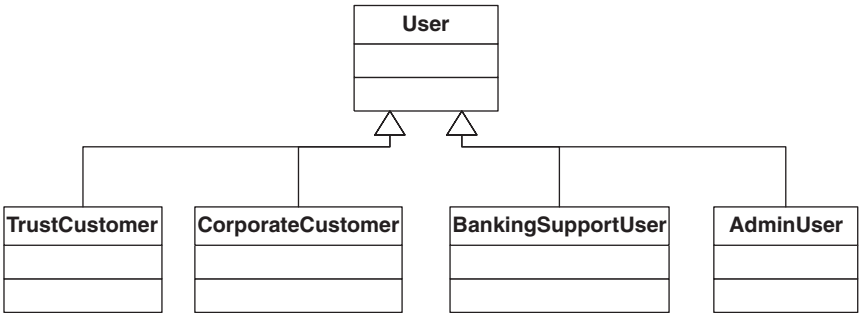


Business Logic Layer

Objects in the business logic layer combine data with business rules, constraints, and activities. Business objects should be separated from DAOs, VOs, and the deployment layer, such as enterprise beans, to maximize the possibility of reuse. Business objects often use and coordinate the activities of multiple data access objects.

Business objects should be deployment independent and self-contained. Any Java Naming and Directory Interface (JNDI) resource (e.g., database connections) that a business object needs to function should be provided by its deployment wrapper. This allows business objects to be redeployed (or

Figure 5.5: Layered Initialization Pattern Example



republished, if you will) as anything you would like, including enterprise beans, RMI services, CORBA services, Web services, applets, and applications.

Some developers add the BO suffix to business object names, but this is not a technical requirement.

Common Patterns

Layered initialization is a pattern you will commonly use when you have different varieties of the same object. For example, most applications have different types of users. As shown in figure 5.5, you might have trust customer users, corporate customer users, banking support users, application administrator users, and so on. All these users share commonality but also have aspects that are unique.

When the same business object might have to produce different outputs or use different inputs, you will most likely use the adapter pattern. Consider the example shown in figure 5.6, a reporting business object that has several different delivery mechanisms—e-mail, printing, Web site publishing—but all other processing is the same. Having an adapter to represent the input consolidates a lot of code.

Like the adapter pattern, the strategy pattern is used when the activities of a business object are likely to vary according to the context. However, the adapter pattern leaves the activities of a class constant while dynamically varying its inputs and outputs, and the strategy pattern makes the activities of a class dynamic while using constant inputs and outputs. It's largely a difference in perception of "who's the client." Figure 5.7 illustrates the strategy pattern.

Figure 5.6: Adapter Pattern Example

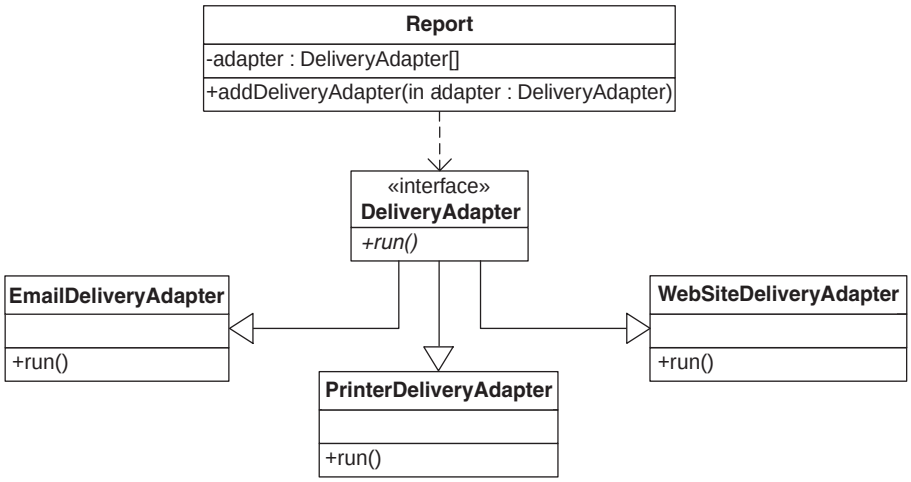
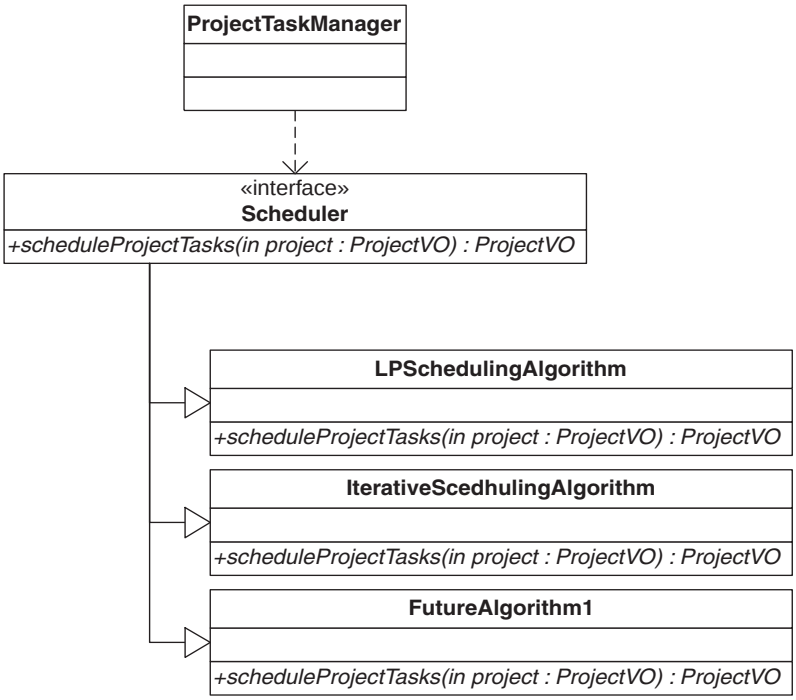


Figure 5.7: Strategy Pattern Example



Deployment Layer

Objects in the deployment layer—called deployment wrappers—are the heart of J2EE architecture. Deployment wrappers publish business object functionality to Java classes that could be on separate machines (including anything in the presentation tier). Examples of deployment wrappers include enterprise beans (e.g., session beans, message-driven beans), RMI services, and Web services. CORBA services are also deployment wrappers but are not considered a traditional part of J2EE architecture.

I consider entity beans to be DAOs, not deployment wrappers. Although technically you can call entity beans from remote clients, doing so slows performance and is therefore not recommended. Calls to entity beans are usually consolidated in session beans. This practice is known as a session bean façade.

Use enterprise beans to publish business logic. Some developers consider enterprise beans to represent business logic. Although it is technically possible to consolidate what I call business objects and enterprise beans into the same class, I don't recommend it. Keeping the business logic separate gives you complete deployment flexibility. You can deploy the same business object as an enterprise bean, message-driven bean, a CORBA service, a Web service, or even a client application with no changes to the underlying business object. Separating the business logic into deployment-independent classes does create additional classes, but the classes are less complex.

Document a deployment wrapper as one class in your object model, even though it's composed of several classes. Take a stateless session bean example. A session bean has a client proxy, an interface, a bean, and a home object implementation. With the exception of the home implementation, all these classes will contain the same method signatures. There's no value in tediously documenting every piece of the deployment layer.

Choosing Deployment Wrappers

You can have multiple deployment wrappers for the same business object. For example, it is not uncommon to create a message-driven bean and a Web service deployment for a customer object.

Each type of deployment wrapper has its advantages and disadvantages. Choosing which type of deployment wrapper to use is difficult if you haven't substantially completed use-case analysis and prototyping (chapter 3) or defining application interfaces (chapter 4). Table 5.3 summarizes the differences between the types of deployment wrappers.

Table 5.3: Features of Deployment Wrapper Types

Feature	EJB	Web Services	Messaging/ JMS	RMI	HTTP	CORBA
Caller platform requirements	Java-compliant only	Any	Any	Java-compliant only	Any	Any
Communication method supported	Synch. only	Both	Both	Synch. only	Synch. only	Synch. only
Coupling	Tight	Loose	Loose	Tight	Loose	Loose
Transaction support	Local and JTA	Local and JTA	Local	Local	Local	Local
Requires J2EE container?	Yes	No	No	No	No	No
Supports clustering for scalability and availability?	Yes	Yes	Yes	No	Yes	Yes

Faced with deciding which deployment wrappers to use and which business objects to publish, ask yourself the following questions; your answers will help you navigate through the decision-making process.

Is your business object called from applications that aren't Java?

This functionality is commonly deployed as a Web service. Web services completely eliminate any language-to-language restrictions by using a common message protocol, UDDI. Because Web services don't represent a programmatic coupling, they need little deployment coordination. Drawbacks to using Web services are that the standards are still being refined. It's a newer, and thus less mature, technology.

Additionally, if the external application is written in a language that supports CORBA and your organization has purchased an ORB supporting that language, you can deploy this functionality as a CORBA service. The CORBA interface is the lowest common denominator so that a wider array of languages can be supported. Supported CORBA languages include C/C++, COBOL, Java, Lisp, PL/I, Python, and Smalltalk.

If your messaging vendor supports JMS and also has a native API for the foreign application platform, you should be able to deploy this functionality as a message-driven enterprise bean.

Is your business object likely to be used by dynamic HTML page constructs, such as servlets and JSPs?

These business objects should be published with a deployment wrapper instead of being used directly by the presentation layer. This allows you to keep the presentation layer, which is typically hard to debug, less complicated. It also allows you to keep the business logic layer deployment generic.

Functionality supporting JSPs and servlets is commonly implemented as a session bean. For best performance, use stateless rather than stateful session beans if the business rules of your application allow it. Avoid using entity beans directly because doing so greatly increases the number of network transmissions and gravely impacts performance.

Does your business object receive and process messages via JMS?

Functionality supporting JMS message receipt and processing is commonly implemented via a message-driven enterprise bean. Although the JMS standard was created and refined in the last few years, messaging technology has existed for more than a decade. Most messaging technology vendors have implemented the JMS interface. Messaging is good for transmitting information. This technology is designed more to guarantee delivery than to issue a subsecond response time.

Does your business object require two-phase commit functionality?

If it does, deploy the business object as a session bean. Two-phase commit functionality requires JTA and is provided by J2EE containers. Web services or servlets running from within a J2EE container will have access to JTA, but these deployments may be used in environments that don't support JTA.

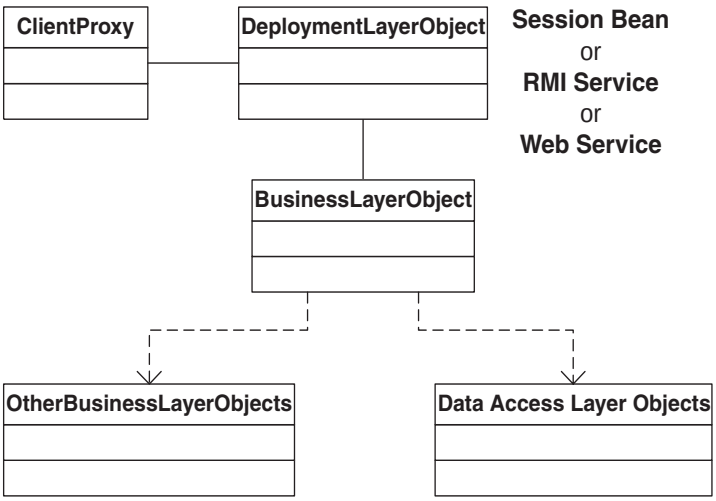
Common Patterns

The pattern commonly used for deployment wrappers is a combination of the session façade pattern and the proxy pattern. This combination (or slight variations thereof) works for all the deployment wrappers that I can think of. The session façade pattern has been especially popular with EJB deployments, but the concept is valid for most types of distributed objects.

One of the primary objectives of the session façade pattern is to minimize network traffic. Figure 5.8 illustrates how you can effectively combine the session façade and proxy patterns.

Although not required by the session façade pattern, I like to provide client proxies for my distributed services, such as enterprise beans, RMI services, and CORBA services. This makes objects in the deployment layer

Figure 5.8: Session Façade Pattern with Proxy Pattern



easier to use because it eliminates the need for a caller in the presentation layer to care about wrapper-specific deployment details. Figure 5.9 depicts enterprise beans deployed in a session façade pattern.

Presentation Layer

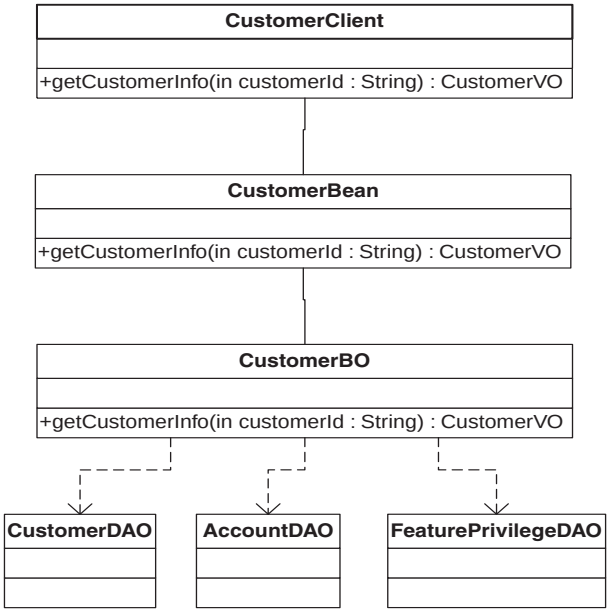
The presentation layer is the section of the application responsible for everything end users physically see in the user interface. Various deployment wrappers provide functionality to the presentation layer.

J2EE developments support HTML/Javascript interfaces and applet interfaces. Most applications provide HTML. J2EE applications produce HTML interfaces by using a combination of static HTML pages and dynamically generated content via servlets and JSPs. There are several good texts for servlets and JSPs (e.g., see Hall, 2000; Hunter and Crawford, 2001). Applets can be used for applications that require advanced controls (actions on mouse-overs, drag-drops, sophisticated interactive display controls, etc.).

The presentation layer uses deployment wrappers exclusively. The reason is that the presentation might not execute on the same host as the business logic and will require distributed services. Although it is technically possible to run the servlet engine on the same host as the J2EE container, some organizations prefer not to.

Most organizations use a variant of the model-view-controller (MVC)

Figure 5.9: Session Bean Deployment Example



pattern for the presentation layer. The MVC pattern is a natural choice for Java because Swing uses the MVC pattern as well. The MVC pattern consists of three parts: (1) the **model** tracks session state and relevant session information, (2) the **controller** interprets all URLs and directs changes to the appropriate model if necessary, and (3) the **view** presents information in the model. In a J2EE world, typically the model is a combination of deployment wrappers (enterprise beans, Web services, and RMI services), the controller is a servlet, and the view is a JSP.

The most popular implementation of the MVC pattern designed specifically for J2EE platform user interfaces is Struts. An open source product from the Apache Jakarta project (http://jakarta.apache.org/struts/), Struts provides a generic, configurable servlet controller that supports J2EE viewers and models. Figure 5.10 is a diagram of the basic parts of Struts and how it fits the MVC pattern.

The classes depicted in the figure (plus an XML configuration file) are the basics needed to implement Struts. Although Struts has many more classes, a major focus of this book is to shorten the learning curve for new architects, so I stick to the basics. Readers desiring a more in-depth knowledge of Struts should see Spielman (2003).

In a Struts world, there are several components that the developer provides. First, a developer-provided XML configuration file (struts-config.xml) tells the `ActionServlet` which `Action` to invoke and which JSP to forward to based on parameters in the URL. The `ActionServlet` class from Struts is serving as the controller for the MVC pattern, and JSPs serve as a view. You will read more about the configuration format and features available for Struts in chapter 14.

Second, the developer optionally provides extensions of `ActionForm` to validate user input. If errors are found, Struts navigates the user back to the URL specified in the configuration file. Your JSPs are responsible for figuring out if there are errors to display and physically displaying error messages.

By the way, the Struts `ActionForm` is also intended as a mechanism to collect user-entered data from a Web page. Use of this feature is optional. If you do use it, any page that has user entry will require an `ActionForm`. This is a Struts feature I rarely use.

Third, developer-provided extensions of `Action` can call objects in the deployment layer, such as enterprise beans, Web services, RMI services, and so on. Calls to the deployment layer initiate some type of process within your application or retrieve data to be displayed to the user. The displayed data are put on the session (just as in a custom servlet). Your JSPs will retrieve any information needed for display from the session.

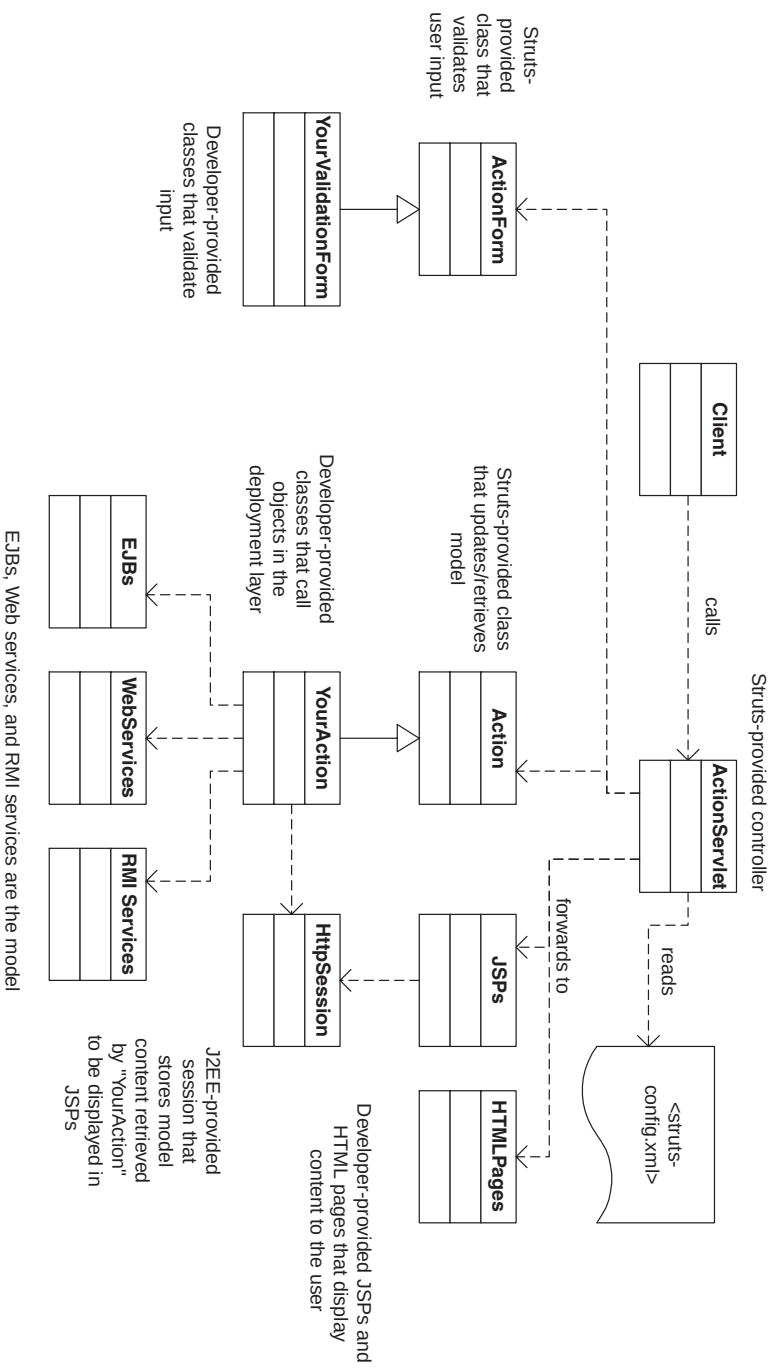
Fourth, the developer provides JSPs and/or static HTML pages to display to the user. The `ActionServlet` forwards the request after the `Action` has completed. Don't worry, Struts does provide a way to dynamically change the forward.

Struts has the advantage of giving you controller functionality you would otherwise have to build, and it's popular, which means many developers already know it. Its chief disadvantage is that it's complex and not the easiest creature to debug when problems arise. Despite its complexity, I usually opt for using it. I'll provide a cheat sheet for getting up to speed on Struts in chapter 14.

Architectural Component Layer

Ideally, all general utilities and components would be provided natively in the Java Development Kit (JDK). But in reality, a gap exists between what the JDK provides and the capabilities applications need. Third-party utilities, APIs, and packages fill a significant portion of this gap, but they do not have complete coverage. Your object-modeling activities probably identify a need for architectural components and utilities that don't yet exist in open

Figure 5.10: MVC Pattern as It Appears in Struts



source form or otherwise. These common components are candidates for sharing across the enterprise.

Whenever possible, use existing architecture utilities as opposed to building them from scratch. Look first to the open source community (it's free and sure to fit in any organization's budget). If you do opt to use an open source package, make sure that you download a copy of the source for the version that you're going to use. If you have production bugs involving an open source package, having the source makes the investigation easier.

Be sure to perform technical feasibility testing for any open source package you select if none of the development team has experience with it. The earlier you find out the limitations of a package, the more time you'll have to react to those limitations and the less they will impact your project timeline.

Although looking for an open source package is a bit like looking for a needle in a haystack, you can shorten the search by starting at the Open Source Developer's Kit site (http://www.osdk.com/). This site surveys and categorizes the most popular open source packages.

Architectural components should be kept application generic for two reasons. First, if they are important enough to share across the enterprise (or perhaps across the Java community as an open source package), they'll be easier to share if they don't have hooks into a specific application. If someone creates an open source package to do this in the future, your homegrown utility will be easier to obsolete.

At this point, you'll want to identify the components and classes that your application will commonly use and leave the details to the developers. The abstract nature and complexity of architectural components can lead to analysis paralysis for many development groups. Specify and model the capabilities and the objects the application will interact with, but leave the details to your most advanced developers.

Further Reading

Alur, Deepak, John Crupi, and Dan Malks. 2001. *Core J2EE Patterns: Best Practices and Design Strategies.* New York: Prentice Hall.

Hall, Marty. 2000. *Core Servlets and JavaServer Pages (JSP)*. New York: Prentice Hall.

Hunter, Jason, and William Crawford. 2001. *Java Servlet Programming*, 2nd ed. Sebastopol, CA: O'Reilly & Associates.

Spielman, Sue. 2003. *The Struts Framework: Practical Guide for Java Programmers.* Boston: Morgan Kaufmann.

6

Creating the Object Model

The technical architect is typically responsible for leading the application design process. In this role, the architect is as much a facilitator as an application designer. This chapter shows you ways to utilize use-case analysis along with the layered approach described in the previous chapter to construct effective designs for J2EE applications and document them in an object model. Along the way, I'll share some techniques for leading a group design process.

Finished use cases are essential for effective application design. Without them, the object-modeling sessions will produce more questions than answers. If you do find vague points or unaddressed issues in the use-case documentation, apply the technique of making assumptions to fill the gaps. Document the assumptions you make so you can confirm them with the business side later. Your use-case analysis is more likely to be incomplete when you're working on a large project.

Object-modeling exercises are applicable for the Java portions of a J2EE application that are custom written (e.g., not provided by a software vendor). Modeling of third-party software components should be limited to those classes directly called by your application.

The technical architect is responsible for facilitating design discussions, which should involve the business logic and presentation-tier developers

in the design process. Because of its subjective nature, the object-modeling process can be a large source of frustration and a project bottleneck. I've seen some projects where the architect produces the object model privately and then attempts to coerce other developers to follow. Although a lone modeler may be able to produce an object model more quickly than a team, the model is more likely to contain errors and omissions, and only the architect understands and is loyal to it.

When the model is produced privately, it doesn't get substantive developer support. After all, if developers can't understand or agree with a plan, they certainly won't want to follow it. Many developers react to architect-produced models by ceasing to contribute anything meaningful to the project from that point. The architect may want to "draft" a model as a way to move the design process along, but developers should clearly understand that this draft is presented solely to prompt discussion and is entirely subject to change.

As a practical matter, when I need to get meaningful support from multiple developers, I go through the longer exercise of forming the plan collectively. Developers who have had input to a model and repeated opportunities to suggest enhancements to it are going to be more enthusiastic about implementing it.

Appoint a scribe for all modeling sessions. It's difficult to facilitate design sessions and take accurate and complete notes of the items discussed at the same time. As the architect is usually leading the session, one of the developers should act as scribe, perhaps on a rotating basis. After the design session, the scribe is responsible for updating the object model to reflect changes discussed in the session.

The remainder of this chapter presents a framework you can use to guide development staff through the object-modeling process. The first step in the process is to identify the major objects in your application using the use cases as a guide. Next, you refine these objects into classes, determine how they interact, and identify attributes and methods. Throughout the chapter, I show you how to streamline the process and avoid those annoying bottlenecks.

Identifying Objects

Identify the most important constructs. Nouns in use cases are generally good candidates for classes. Thus a good way to start identifying objects is by reading the use cases and extracting a list of all the nouns. (You can ignore *system* in the beginning phrase, "The system will," because it's merely part of the use-case format.)

At this point, you should interpret my use of the word *object* loosely. In the early stages of development, it's impossible to know enough about these objects to understand exactly which classes you will derive from them. Note that my use of the term *object* differs from some texts that use the term to refer to an instance of a class.

Don't bother with attribution or relationships at this stage. Attribution and relationships are important, but identifying them too early will bog you down in too much detail and will throw the team into frequent tangents. For now, try not to be concerned about process and focus instead on data organization.

As objects are identified, record persistence requirements. Some classes will represent data that your application has to store, usually in a database, and are said to be **persistent**. In fact, persistent objects frequently appear as entities in the data model. I often record objects with persistent data as entities in the data model as they're identified in the object model. I discuss data modeling in detail in chapter 7.

Objects identified at this stage are high level. You will further refine and expand them later in the process, using objects to determine specific classes.

Object Identification Example

Let's use an example paraphrased from a reporting system I once implemented. The team defined the following uses cases:

- The system will provide an interface that will accept report template definitions from an existing MVS/CICS application. A report template consists of an ID, a name, a list of parameters required to run the template, and a list of data items produced by the template.
- The system will allow application administrators to control the report templates that users belonging to a trust customer organization can run.
- The system will run reports at least as fast as its predecessor system did on average.
- The system will restrict reported data for all trust customer users to that of the trust customer organization to which they belong.
- The system will allow banking support users to execute all report templates using data from any trust customer organization.

Looking at the nouns in the use cases in order (ignoring *system*, as mentioned earlier) gave us the list that appears in the first column of table 6.1.

Table 6.1: Object Identification Example

Noun (fromuse case)	Object
Interface	`ReportTemplateInterface`
Report template	`ReportTemplate`
List of parameters	`ReportTemplateParameter`
Data item	`ReportDataItem`
Application administrator	`ApplicationAdministrator`
Trust customer organization	`TrustCustomerOrganization`
Trust customer user	`TrustCustomerMember`
Reported data	`Report`
Banking support user	`BankingSupportUser`

Next we rephrased the nouns to make them self-contained object names, as shown in table 6.1. By self-contained, I mean that object names shouldn't depend on context to provide meaning. For instance, *interface* from the first use case became `ReportTemplateInterface` and *list of parameters* became `ReportTemplateParameter`. The fact that our use case referred to a "list" of parameters was documented as a relationship. The more descriptive the object name, the better. All the objects were persistent except `Report-TemplateInterface`. (Note that the word *interface* in this use case refers to an application interface and may not imply use of a Java interface construct.)

Three types of users appear in the list: application administrator, trust customer member, and banking support user. When we got to attribution, we recognized that there was another object, User, with different subtypes. Inheritance relationships like this are easier to recognize when it comes time for attribution, so let's leave the object list as it is for now.

An alternative to merely identifying nouns is to do the data-modeling exercise first. All identified entities are good object candidates. Many of the objects we identified in this example would make good entity candidates as well. See chapter 7 for details.

Some of these objects were implemented as classes in multiple software layers in the application. This process is the focus of the next section. Defining and illustrating these layers was described in chapter 5.

Turning Objects into Classes

Once you have identified major objects in the application, you need to refine those objects into classes and organize them in a framework.

After identifying objects, you need to identify which layers they pertain to. It is common for objects to have multiple roles. Any object you have identified as playing multiple roles (e.g., manage data access, implement business rules, etc.) must get counterparts in multiple layers. For example, `ReportTemplate`, `ReportTemplateParameter`, and `ReportDataItem` from table 6.1 had persistence requirements as well as requirements as business objects. Therefore, they appeared as classes in at least the data access layer, the business object layer, and the value object layer.

Define separate classes for each object in each layer. If you define the same class for each of these roles, the classes get too large to effectively maintain and you lose all the benefits from software layering.

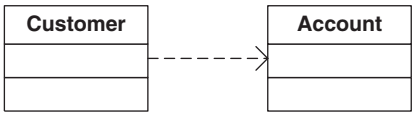
From the object identification example, consider object `ReportTemplate`. Class `ReportTemplateDAO`, residing in the data access layer, was responsible for reading and writing template information using a relational database. `ReportTemplateVO` in the value object layer described all characteristics of a report template (e.g., its name, data type, display length, etc.). `ReportTemplateBus` in the business logic layer coordinated and enforced all rules for creating a new report template.

Determining Relationships

A **relationship** describes how different classes interact. You can determine relationships after you've identified individual classes. The UML literature documents several categories of object relationships. Most applications only need the four types described in the following paragraphs.

Dependency (uses) relationship documents that one class uses another class. At a code level, *using* another class means referencing the class in some way (e.g., declaring, instantiating, etc.). Because this relationship type is strikingly similar to the association relationship type, I usually ignore the difference between the two types as unnecessary complexity. The relationship in figure 6.1 is read as "`Customer uses Account`."

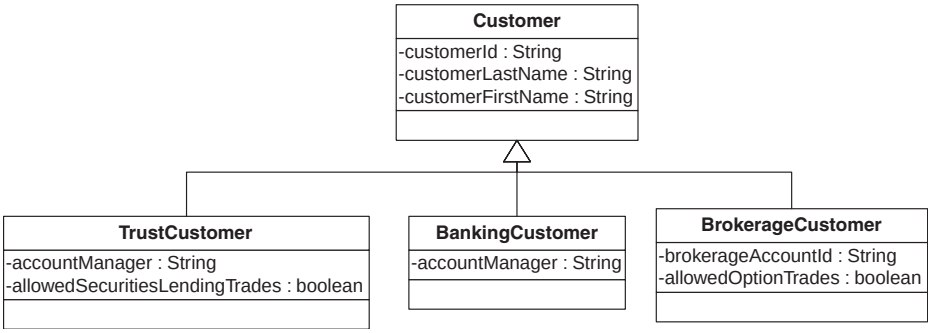
Figure 6.1: Dependency Relationship Illustration



Generalizes (extends) relationship documents that one class extends, or inherits the characteristics of, another class. Most medium to large applications have a handful of this type of relationship.

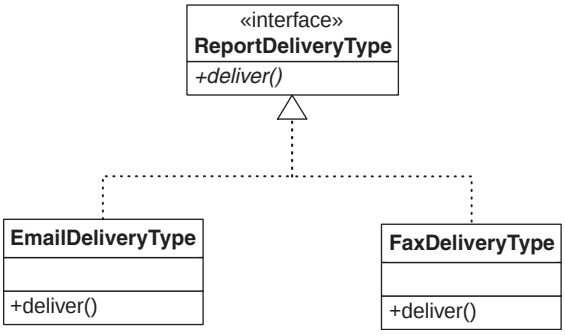
An extends relationship is denoted by a solid line with a hollow arrow. The relationship in figure 6.2 is read as "`TrustCustomer extends Customer.`" The attributes of Customer will be available and usable in all children.

Figure 6.2: Extends Relationship Illustration



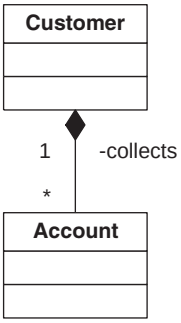
Realization (implements) relationship documents that a class implements an interface. An implements relationship is denoted by a dotted line with a hollow arrow. The relationship in figure 6.3 is read as "`EmailDelivery` implements `ReportDeliveryType.`"

Figure 6.3: Implements Relationship Illustration



Aggregation (collects) relationship documents that a class collects multiple occurrences of a class. A collects relationship is denoted by a solid line with a diamond next to the class doing the collecting. The relationship in figure 6.4 reads "`Customer` collects `Account`."

Figure 6.4: Collects Relationship Illustration



Don't document formal relationships to value objects. VOs have so many relationships that if they are documented, the model becomes unreadable and unusable. After all, the purpose of creating the class diagram is to make the application easier for the team to understand and implement. VO relationships are easy for developers to figure out from the method signatures anyway.

Identifying Attributes

Attributes are fields that a class contains. At a code level, attributes are instance-level variable declarations. Most attribution occurs with VOs, with other object types receiving little attribution.

Ideally, attributes should be base, not derived. A **base attribute** is atomic—that is, its value is not derived from the value of other elements or the result of a calculation. Conversely, a **derived attribute** is made up of the values of other elements. For example, consider a `CustomerVO` class that has `firstName`, `lastName`, and `fullName` attributes. The attribute `fullName` is derived because it is made up of the first and last names.

Avoid declaring derived attributes. Derived attributes, like `fullName` mentioned in the previous paragraph, only give you more to maintain. If a customer changes his or her last name, the values of two attributes need to change. Instead of making `fullName` an attribute, it would be better to create a method, such as `getFullName()`, that does the concatenation.

Identifying Methods

Methods are where the action is. Methods are invoked primarily when a user does something, when something that was scheduled occurs, and when something is received from an external interface. A common way to identify methods is to analyze each event and document the methods needed along the way. During the course of identifying the methods you'll need, you'll usually identify new classes.

Starting with user actions, I use screens from the prototype to drive method identification. Consider an application login as an example. Most applications customize the main screen based on a user's identity and preferences. For the moment, assume that authorization and authentication is provided by the enterprise architecture and not at an application level. Further assume that you need to invoke something that will get user specifics from the security package, invoke something to retrieve that user's preferences regarding your application, and display the main application page.

If you use Struts, you'll need some kind action class that can get user specifics from the security package, using the package to look up customer preferences (which will be used to generate the main page). If you haven't yet identified an action class to do this, add it to the model. Ignoring the security package itself, since it's out of scope, you then need a method somewhere that allows the user preference look-up.

Remember from our discussion of software layering in chapter 5 that classes in the presentation tier use deployment wrappers rather than using DAOs directly. Most applications identify a user object (or something similar) that manifests into a DAO class, a VO class, a business object, and a deployment wrapper for an enterprise bean. It would be logical to add a

`getUser(String userId)` method to our `User` deployment wrapper. It would also be logical for that method to return a `UserVO`, which includes information about preferences, as a result of the call. As a shortcut, assume that this method is added to the client proxy and all necessary components (controller interface and bean) of the deployment wrapper.

Pass and return VOs instead of individual data items. Doing so can increase performance when you use enterprise beans as a deployment wrapper. By reducing the number of methods that need to be changed, passing and returning VOs also reduces the amount of code that you'll need to change when you enhance the application.

Deployment wrappers typically don't contain application logic but pass through to a business object. This method will have to be added to the business object as well. The deployment wrapper, when `getUser()` is called, instantiates and calls a similar method on the business object, `UserBus`.

The physical look-up of the user preferences is usually delegated to at least one DAO. It would be logical to have a `UserDAO` that has a `select()` method on it that would do the look-up.

By now, you're probably wondering why you don't just have the `Action` class invoke the DAO directly. It would appear to be simpler because it would cut out several layers of classes that, in this example, appear to be adding little to the functionality of the product. Technically, you could have had the `Action` class invoke the DAO directly, but doing so would have greatly added to the complexity of the `Action` class, which isn't easy to debug, and would have eliminated the insulating benefits of software layering, as discussed in the last chapter.

Figure 6.5 illustrates an object model for the example just discussed.

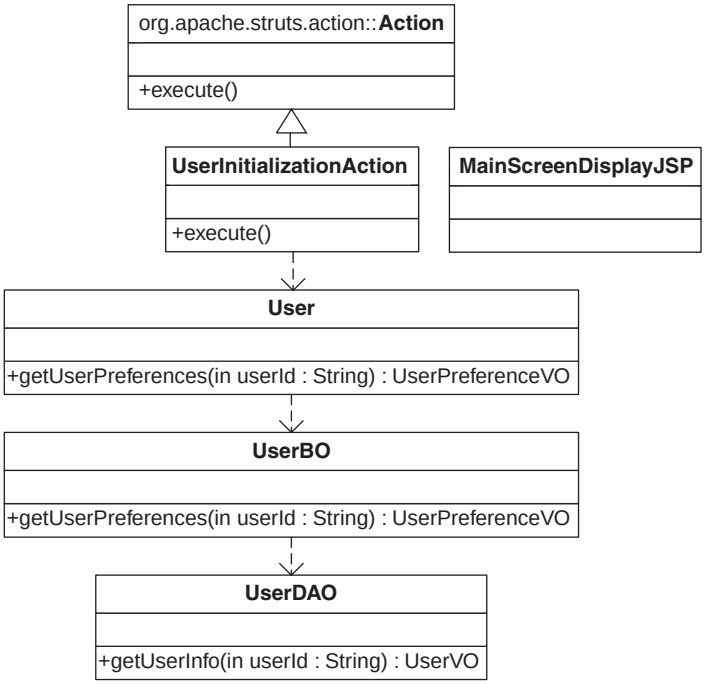
Shortcuts

Over the years, I've adopted several shortcuts that decrease the time and effort required in the object-modeling phase.

Document deployment wrappers as one class. How deployment wrappers (e.g., session beans) are structured is standard, and there is no value in documenting it repeatedly. For example, model an EJB as one class instead of breaking it up into its three or four component parts that all have the same signatures anyway.

Assume that all documented attributes will have of accessors (get methods) and mutators (set methods). This simplifies the model and eliminates

Figure 6.5: Object Model Example



a lot of boring busywork. It also makes the model easier to read and thus more useful. For example, if you document that a `CustomerVO` class has a `lastName` attribute, you should be able to assume the existence of `getFirstName()` and `setFirstName()` methods without explicitly documenting them in the model.

Omit relationships to objects in the JDK. Including these relationships adds little value but a lot of complexity, making the model hard to read. For example, many attributes in the JDK are strings. Technically, any class containing a string attribute should have a relationship to the `java.lang.String` class. In a typical application, documenting these types of relationships would add hundreds or thousands of relationships to your model and provide no benefit.

Forget about generating code from UML. My experience is that it's usually a net time loss. The minute level of detail you have to type in the model usually exceeds the time it would take to code it using any integrated development

environment (IDE). On the other hand, generating UML from existing code can be an extremely valuable and easy to do if your IDE supports it.

Don't attempt to micromanage the coding when determining methods. Leave some work to the developer. If the development team starts documenting private methods, it's going too far.

Architect's Exercise: ProjectTrak

Let's apply these concepts to the ProjectTrak example used in previous chapters. The following use case is from ProjectTrak:

- The system will allow users to define, edit, and display project tasks. A project task has a name, an estimate (in hours), percent complete, one assigned personnel resource, any number of dependent tasks, and a priority (high/medium/low).

Starting with the presentation layer, let's assume that the prototype informs us that there is one page for task definition, edit, and display, with a push-button control for the Save capability. The one display must be dynamically generated, so it can't be a static HTML page. We'll make it a JSP and call it `TaskEditDisplayJSP` in our object model.

Further, let's assume that there is a control on another page (out of scope of our example) that causes the task display/edit page to come up either with information from an existing task or blank (in the case of a new task). And let's assume that we're using Struts.

We need an action to look up the existing task information and store it on the session for the `TaskEditDisplayJSP`. If this is a new task, our action needs to initialize the appropriate variables on the session for the `TaskEditDisplayJSP`. This action, like all actions, will extend `org.apache.struts.action.Action`.

The action, which we'll call `TaskDisplayAction`, should call something in the deployment layer to get task information. Because this is a J2EE application and we don't need to support anything but Java clients, let's make this deployment layer object a session bean (we can choose between stateless and stateful later). We'll call it `ProjectBean`. As discussed previously, let's create a client for the bean to make it easier for presentation-tier developers to call. We'll call that client proxy `ProjectClient`.

Further, we need a method on the bean and client that retrieves project task information. Let's identify that method as `getProjectTask()`. As with

all get-type services, we need to pass an argument that identifies which task to retrieve, and we need a VO object to represent a project task returned.

Our use case has enough detail to identify and define the `ProjectTaskVO` object with all the attributes listed from the use-case definition. As discussed in the "Shortcuts" section, we'll list only the attributes in the model, not the accessors and mutators. We can assume that the object will have `getTaskId()` and `setTaskId()` methods, even though we won't list them in the model. We also won't document the relationships between `ProjectTaskVO` and all the objects that use it because it would make the model unreadable and not useful.

`ProjectBean` will need to get project task information from an object in the business logic layer. The `ProjectTaskBO` object in the business logic layer needs to access a DAO to look up the task.

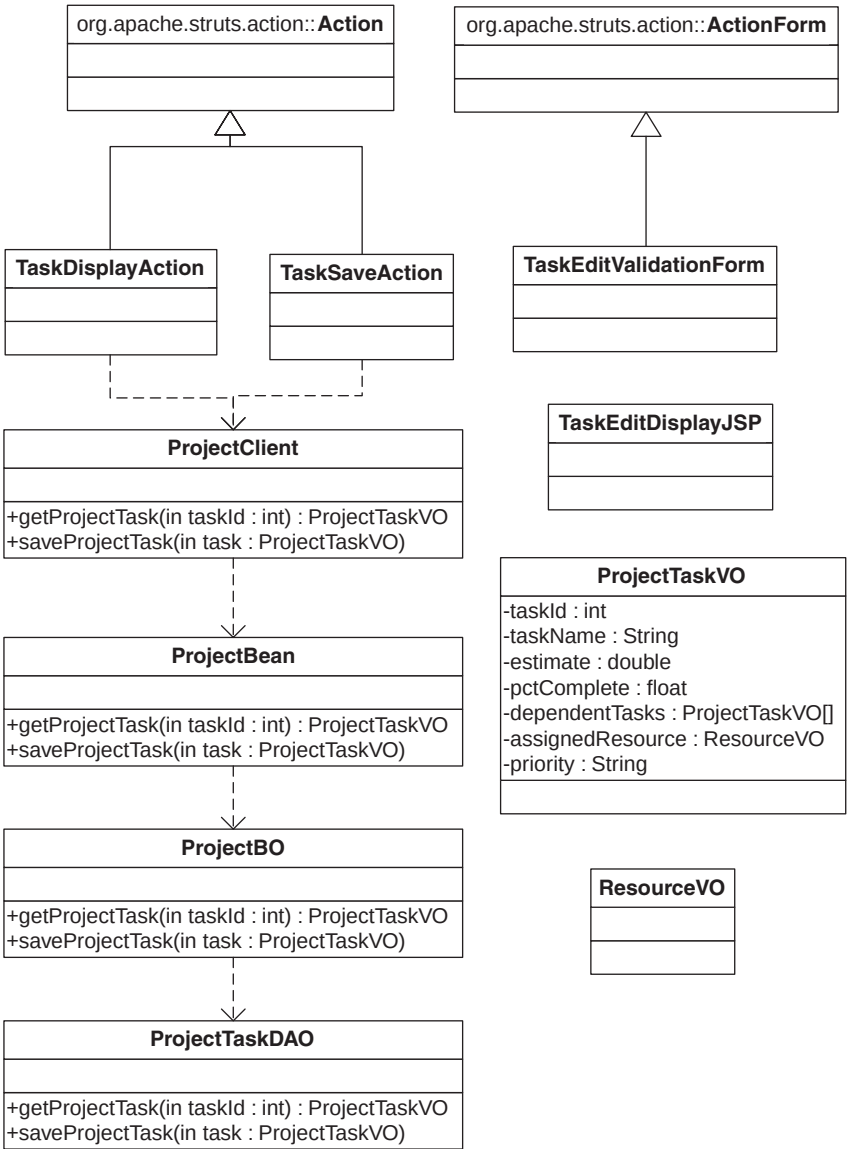
After users input or change task information, they have the option to save. We need an action in the presentation layer to initiate the save. Let's call it the `TaskSaveAction`. The `TaskSaveAction` should include something in the deployment layer to initiate the save. We also need a validation form to make sure users entered valid data. This form extends `org.apache.struts.action.ActionForm`. Let's call it `TaskEditValidationForm`.

Following much the same logic that we used to identify the `getProjectTask()` methods in our client proxy, enterprise bean, business object, and data access object, we can identify a `saveProjectTask()` method to initiate save processing at all these levels.

Figure 6.6 is an object model for everything we identified in this example.

Note that this model has something that's very extensible. If we need to support .Net clients at some point, we can create a Web service deployment for the `PROJECT_BO`, and all other functionality remains the same. If we change our data storage mechanism, we can do so without affecting any other tier. The layering concept used here provides us insulation against change.

Figure 6.6: ProjectTrak Object Model



Further Reading

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Grand, Mark. 2002. *Java Enterprise Design Patterns*. New York: John Wiley & Sons.

Johnson, Rod. 2002. *Expert One-on-One: J2EE Design and Development*. Indianapolis, IN: Wrox Press.

Taylor, David. 1990. *Object-Oriented Technology: A Manager's Guide*. Reading, MA: Addison-Wesley.

7

Creating the Data Model

Most books about J2EE skip the data-modeling and database design phases of development. But I have found that both steps are critical to the success of a J2EE application. As part of the development team, a technical architect should have a basic knowledge of data modeling. Further, in some companies, the architect is responsible for designing the application database. Just as object-modeling exercises help produce good code, data-modeling exercises help produce good database designs. (I have to admit, however, that the many years I spent as a database designer and administrator may have prejudiced my views as to the importance of data modeling.)

In addition, I find data-modeling concepts useful in designing XML document formats, such as DTDs and schemas. Applying data-modeling concepts to XML document design is a bit unconventional. The thought process behind deciding if a data item is an element or an attribute is similar to deciding between entities and attributes in data modeling. In addition, one-to-many relationships in data modeling translate directly to the child element concept in XML documents. I'll provide some details and examples in this section to show how you can implement data models as XML document formats.

Although relational databases may someday be usurped by object databases, I don't see any signs of that occurring in today's market. For now, because relational databases are part of most J2EE applications, most technical

architects need to have at least a basic understanding of data-modeling concepts.

If you're more comfortable with data modeling than with object modeling, feel free to take the easier path by doing data-modeling activities before object modeling. All the *entities* (defined in the next section) in the data model are potential identifications of data access objects, business objects, and value objects. Although the two modeling disciplines use different terms, they are quite similar conceptually.

Key Terms and Concepts

An **entity** is something you want to keep information about and thus represents information that persists (i.e., is written to media). Usually, an entity is a noun. Although most entities are implemented as database tables, the terms *entity* and *table* are not synonymous. An entity is purely a conceptual construct, with its closest counterpart in object modeling being a class. Good examples of entities are customer, account, user, customer order, and product.

In a relational database, an entity is implemented as a table. When you implement your data model as an XML DTD or schema, then each entity becomes an element.

An **entity occurrence** (sometimes shortened to occurrence) is an instance of an entity. If you're more comfortable with object modeling, you can think of an entity occurrence as similar to instantiating a class. If you can't resist the urge to equate entities and tables, consider an entity occurrence as a row in a table. And for XML users, an entity occurrence is like an individual element in an XML document.

An **attribute** is a characteristic of an entity. Although attributes can be nouns, they usually don't make sense outside the context of an entity. For example, attributes of a `CUSTOMER` entity could be `CUSTOMER_ID`, `FIRST_NAME`, `LAST_NAME`, `STREET_ADDRESS`, `CITY`, `STATE`, and `ZIP_CODE`. Attributes should be **atomic**—that is, they should be self-contained and not derived from the values of other attributes.

A **primary key** is the one attribute, or combination of attributes, of an entity that uniquely identifies an entity occurrence. For example, `CUSTOMER_ID` would be a good primary key for the `CUSTOMER` entity, and `ACCOUNT_NUMBER` and `ORDER_NUMBER` taken together would be a good primary key for an entity called `CUSTOMER_ORDER`.

Every entry must have a primary key. If, by chance, no combination of attributes uniquely identifies an entity occurrence, make up an attribute to

serve as a key. For example, most manufacturers assign unique identifiers (UPC codes) to their products, but when they don't, you might have to make up product IDs to serve as the primary key.

A **relationship** is an association between two entities. Out of the many types of relationships that exist, three are commonly used: one-to-many, many-to-many, and supertype/subtype. In a **one-to-many** relationship, one occurrence of an entity is associated with possibly multiple occurrences of another entity. For example, a single customer could have multiple accounts or place multiple orders. Often, the entity with a single occurrence is called the **parent**, and the entity with multiple occurrences is called the **child**. Figure 7.1 illustrates a one-to-many relationship.

Notice in the figure that the `ACCOUNT` entity contains the primary key (PK) columns of the `ACCOUNT_TYPE` and `CUSTOMER` entities. Each additional column in `ACCOUNT` is a foreign key (FK). **Foreign keys** are the primary keys of related entities that an entity uses for look-up purposes. The existence of a foreign key is an implicit result of creating a one-to-many relationship. For example, given an occurrence of `ACCOUNT`, related `CUSTOMER` and `ACCOUNT_TYPE` information is easy to determine.

Some data-modeling tools provide for many-to-many relationships between entities. In a **many-to-many** relationship, each entity has a one-to-many relationship with the other. For example, customer orders can contain many products, and each product can be purchased by multiple

Figure 7.1: One-to-Many Relationship

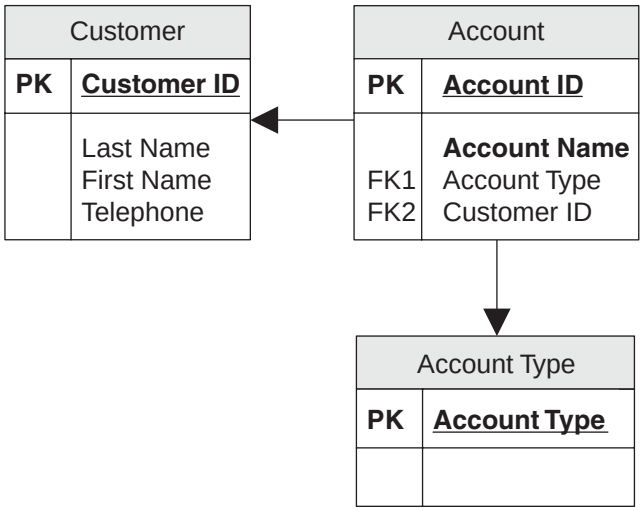
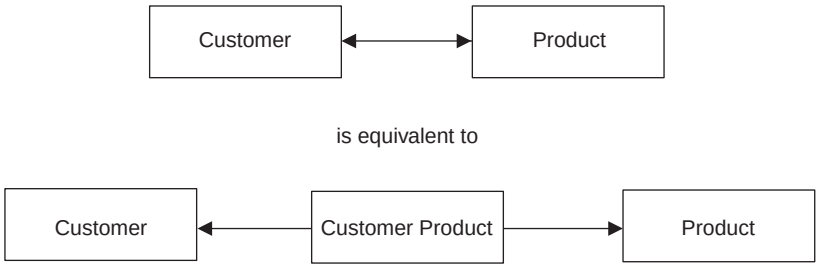


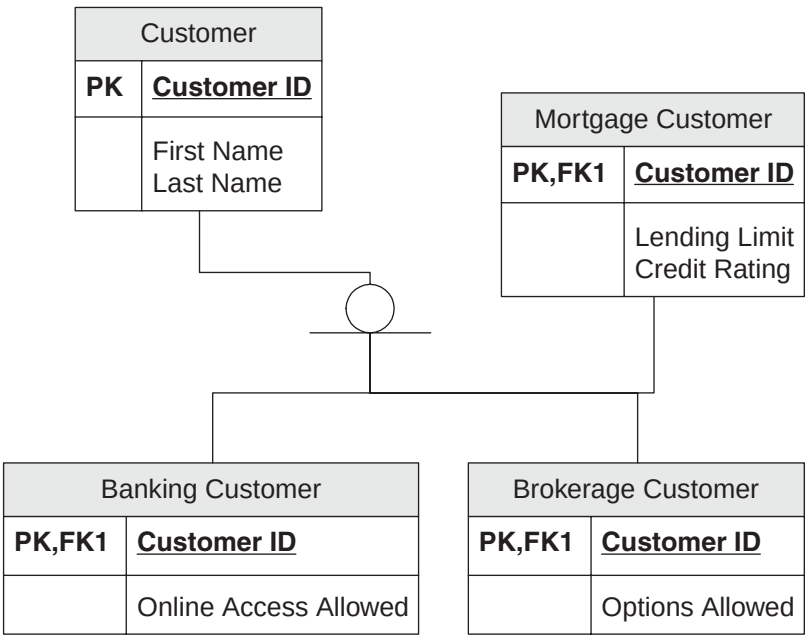
Figure 7.2: Many-to-Many Relationship



customers. It is common to rewrite a many-to-many relationship as two separate one-to-many relationships with a new entity defined as a cross-reference. Figure 7.2 is an example of a many-to-many relationship.

In a **supertype/subtype** relationship, an entity refines the definition of another entity. For example, in banking, a customer entity might be too generic for a bank that has trust customers, private banking customers, corporate customers, brokerage customers, and so on. As shown in figure 7.3, the CUSTOMER entity is the supertype, and the others are the subtypes.

Figure 7.3: Supertype/Subtype Relationship



It is possible to have entities related to themselves. This is called a **recursive relationship**. For example, consider an `EMPLOYEE` entity with `EMPLOYEE_ID` as the primary key. A recursive one-to-many relationship could be used to indicate the manager of each employee. As a result of the relationship, a foreign key, say `MANAGER_ID`, would be used to cross-reference employees with their managers.

Design Practices and Normal Form

Normal form is a set of rules that guide you in identifying entities and relationships. In fact, there are many different degrees of normal form, but in practice, third normal form is the one most frequently used. For that reason, I limit the discussion here to third normal form; if you are interested in other normal forms, I recommend Date (2003).

To qualify for third normal form, entities must satisfy three conditions:

1. All repeating attribute groups should be removed and placed in a separate entity.
2. All nonkey attributes should be dependent only on the primary key.
3. All nonkey attributes should be dependent on every attribute in the primary key.

Suppose the `CUSTOMER` entity has the attributes `ADDRESS_LINE_1`, `ADDRESS_LINE_2`, `ADDRESS_LINE_3`, and `ADDRESS_LINE_4`. Technically, such an entity isn't third normal form because it's a repeating group and violates condition 1. Figure 7.4a illustrates the example of this bad practice, and Figure 7.4b illustrates a possible correction.

Figure 7.4a: Violation of the First Condition of Third Normal Form

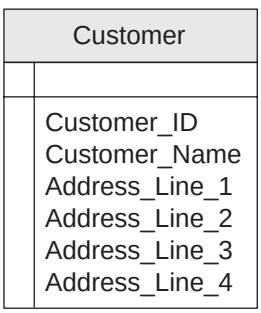
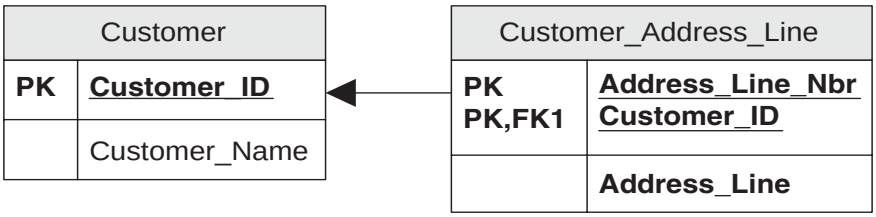


Figure 7.4b: Violation Corrected



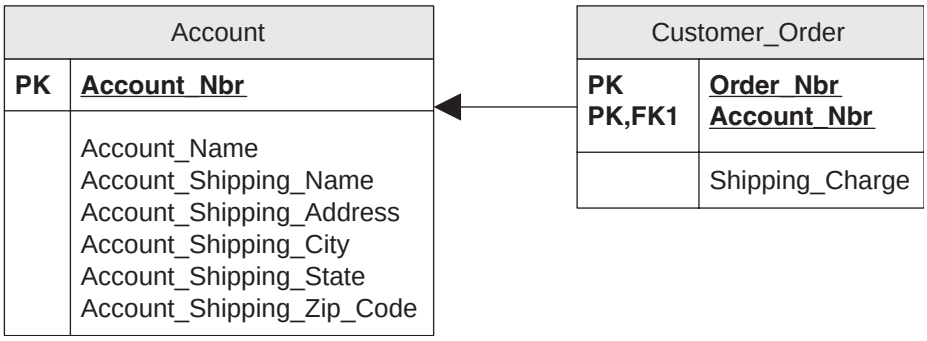
Suppose the ACCOUNT entity contains the attribute ACCOUNT_BALANCE. This isn't third normal form because it violates condition 2. ACCOUNT_BALANCE is fully dependent on outstanding orders, the line items on them, and the payments that have been made—items in other entities. Another problem with ACCOUNT_BALANCE is that it isn't atomic. ACCOUNT_BALANCE is computed based on previous orders and customer payments.

Suppose the CUSTOMER_ORDER entity (which has a primary key that combines ACCOUNT_NUMBER and ORDER_NUMBER) has the attributes ACCOUNT_NAME and ADDRESS_INFORMATION. This technically isn't third normal form because these attributes relate to the account but not the specific order, which violates condition 3. Figure 7.5a illustrates the order violating third normal form, and Figure 7.5b illustrates a corrected version.

Figure 7.5a: Violation of the Third Condition of Third Normal Form

Customer_Order	
PK PK	**Account_Nbr** **Order_Nbr**
	Account_Shipping_Name Account_Shipping_Address Account_Shipping_City Account_Shipping_State Account_Shipping_Zip_Code Shipping_Charge

Figure 7.5b: Violation Corrected



Architect's Exercise: ProjectTrak

Now that we've defined all the key data-modeling terms, let's pull them all together into a real application. We'll continue with the following use case from ProjectTrak:

▲ *The system will allow users to define, edit, and display project tasks.* A project task has a name, an estimate (in hours), percent complete, one assigned personnel resource, any number of dependent tasks, and a priority (high/medium/low).

The first task in the data-modeling exercise is to identify the entities. And the first question we need to answer is: what things are we keeping information about? A project task is the most obvious candidate and alludes to the existence of another entity, a project. Although the use case doesn't give us any information about the characteristics of a project, we should identify it as an entity anyway.

The second sentence of the use case gives a list of characteristics for the `PROJECT_TASK` entity. Those traits that are atomic will be attributes of `PROJECT_TASK`. The name and estimate for `PROJECT_TASK` are definitely atomic, so let's add them to `PROJECT_TASK` as attributes.

`PERCENT_COMPLETE` might (or might not) be an atomic attribute. Another use case for this project states that the product needs to track the time people work on each task. So we could calculate the percent complete for each task by adding up the time people have contributed to the task and dividing that by the hourly estimate. We could argue that the estimate might be wrong and that we might use up all the time estimated for a particular task before completing it. Because the use case doesn't make the issue clear,

let's assume that the PERCENT_COMPLETE is atomic and might not correlate to the time worked toward a task. We'll check that assumption with a business analyst later.

The phrase "personnel resource" is a bit ambiguous. If we only had this use case to consider, we might assume this to mean the name of that resource and model it as an attribute of PROJECT_TASK. However, one of the other use cases mentions tracking skill sets for a resource. That would indicate that a resource is a separate entity with a relationship to PROJECT_TASK rather than an attribute of PROJECT_TASK. We'll go ahead and identify the entity RESOURCE, even though this use case doesn't tell us its attributes. We also know that there is a relationship between PROJECT_TASK and RESOURCE.

The phrase "any number of dependent tasks" indicates that PROJECT_TASK has a recursive relationship (a one-to-many relationship to itself). The children tasks in this relationship are other tasks that must be completed before we can consider PROJECT_TASK complete.

PRIORITY might (or might not) be an appropriate attribute. A data-modeling purist might tell you that a priority is separate from a project task and thus should be modeled as an entity. If we did this, we would need to put a one-to-many relationship between the PRIORITY and PROJECT_TASK entities.

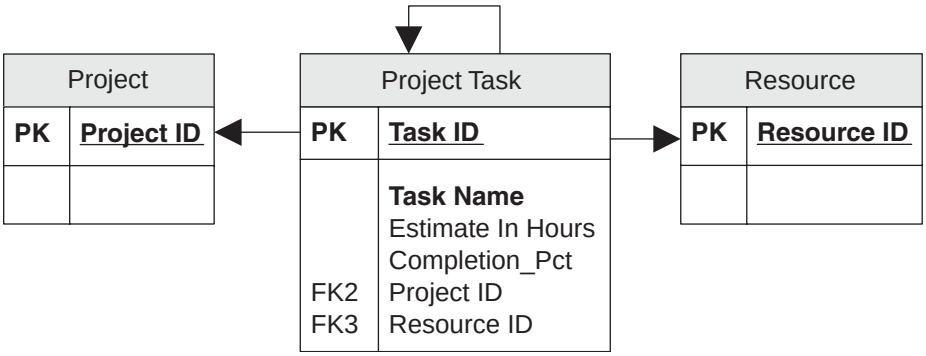
If we had identified a use case that provided for users to define their own priority levels, we would definitely make PRIORITY an entity. However, because the use case only mentions priority as a characteristic of a project task, let's model PRIORITY as an attribute of PROJECT_TASK.

While the use case doesn't specify a relationship between PROJECT and PROJECT_TASK, most people would feel comfortable that there is one. Another ProjectTrak use case states that projects can have multiple tasks. Although we could be accused of jumping the gun a bit, we'll go ahead and draw in the relationship.

Next, we need to identify primary keys for our three entities. At this point, let's assume that we have to artificially construct a key for each. We'll call them PROJECT_ID, TASK_ID, and RESOURCE_ID. Typically, we would record the assumption and revisit it after we've analyzed several other use cases. We may be able to identify a natural key for each entity later in the analysis. Natural keys are preferable to artificially constructed keys because they are more intuitive and understandable. Artificial keys also require additional application logic to generate and maintain.

Perhaps you felt a bit uncomfortable with making assumptions in this exercise. But it's a fact of corporate life: few decisions (including design

Figure 7.6: ProjectTrak Data Model Example



decisions like the ones we made here) are made with perfect information. Try as you might, your use cases will never be 100 percent complete.

Figure 7.6 shows the ProjectTrak data model.

Creating Database Schema Definitions

Typically, database administrators use the data model to create relational database schemas for the rest of the team to use. And most database administrators use modeling tools to do the dirty work. Unfortunately, few open source tools for creating schemas are available. Although the process is a bit more involved than what I illustrate in this section, with the help of a qualified database administrator, you can create schemas using the following central algorithm:

1. *Directly translate each entity into a table.* All attributes of an entity become columns in the table. Explicitly define the primary key in each table.
2. *Assign a foreign key in the child entity of each one-to-many relationship.* Remember, a foreign key is the primary key of another entity that exists so that you can match data in one entity to another. For example, `CUSTOMER_ID` will appear as a foreign key in the `ACCOUNT` table so that you have a way to associate an account with a specific customer using a SQL join.

3 *Rewrite each many-to-many relationship by adding an associative table and two one-to-many relationships.* An associative table has a primary key that is made up of two foreign keys. For example, look back at figure 7.2 to see the many-to-many relationship between CUSTOMER and PRODUCT. This will get implemented by creating a new table (called CUSTOMER_LINE_ITEM, for example) that relates customers to products.

As an illustration, listing 7.1 translates the ProjectTrak data model from figure 7.6 into Oracle DDL.

Listing 7.1: ProjectTrak DDL for Figure 7.6

```
create table Project (Project_ID number primary key);

create table Project_Task
 (      Task_ID number  primary key,
        Task_Name       varchar(50)     not null,
        Estimate_In_Hrs number,
        Completion_Pct number,
        Project_ID      number          not null,
        Resource_ID     number
 )

create table Resource (Resource_ID number primary key);

ALTER TABLE Project_Task
 ADD CONSTRAINT Project_FK
 FOREIGN KEY (Project_ID)
 REFERENCES Project(Project_ID);

ALTER TABLE Project_Task
 ADD CONSTRAINT Resource_FK
 FOREIGN KEY (Resource_ID)
 REFERENCES Resource(Resource_ID);
```

Common Mistakes

Denormalizing the database out of habit. Denormalizing a database means replicating information to avoid look-ups and enhance performance. Consequently, denormalization can introduce maintenance problems if the two copies get out of synch.

In the early days of relational databases, denormalization for performance was a must. However, the technology has advanced to the point where forced

denormalizations are rare. Today, denormalizations are done more out of (bad) habit than for performance reasons.

Dropping database integrity constraints for programmatic convenience. Some developers like to shut off the foreig key relationships between tables. Not using database integrity constraints initially saves the programmer time because it permits invalid inserts, updates, and deletes. But I've found you lose more time than you save because you end up having to fight bugs created by flawed inserts, updates, and deletes. The sooner you catch a bug, the cheaper and easier it is to fix.

Creating XML Document Formats

In addition to their use in database design, data-modeling techniques can easily be applied to designing XML documents. The same data models that database administrators use to create physical database designs also readily translate into XML document formats, such as DTDs or schemas. XML is most often used as a means of communication between applications.

The first step in creating any XML document is to identify the document root. XML documents usually contain lists of things identified in the data model. For instance, a `<customer-update>` document might contain a list of customer-related elements that contain information hat has changed. A `<purchase-order>` document might contain a list of order-related elements describing one or more purchase order contents.

Entities in a data model translate to elements in an XML document. Only implement the elements that are needed for the documents you're creating. Chances are that you don't need all entities translated into elements. Entities that represent small look-up value domains (e.g., `CUSTOMER_TYPE`, `ACCOUNT_TYPE`, etc.) are usually implemented as attributes rather than elements in an XML document.

Attributes of an entity become attributes of the corresponding element. For example, the `<customer>` element from figure 7.1 would have the attributes `customer-id`, `last-name`, `first-name`, and `telephone`.

A one-to-many relationship implies that one element is the child of another in an XML document. Unlike relational databases, a foreign key to the parent element isn't needed because it's indicated by segment ancestry. Ancestry is indicated naturally within the XML syntax. For example, the

<customer> element from figure 7.1 would have an optional <account> child element.

As a more complete illustration, listing 7.2 is a sample XML document for the data model in figure 7.1.

Listing 7.2: XML Documnt Example

```
<?xml version="1.0" encoding="UTF-8"?>
<customer-update>
        <customer       customer-id="C123"
first-name="Derek"
last-name="Ashmore"
telephone="999-990-9999">
                <account account-id="A1"
account-name="Personal Checking"
account-type="checking"/>
        </customer>
</customer-update>

A sample DTD definition for this XML document type follows:
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT account EMPTY>
<!ATTLIST account
        account-id CDATA #REQUIRED
        account-name CDATA #REQUIRED
        account-type CDATA #REQUIRED
>
<!ELEMENT customer (account)>
<!ATTLIST customer
        customer-id CDATA #REQUIRED
        first-name CDATA #REQUIRED
        last-name CDATA #REQUIRED
        telephone CDATA #REQUIRED
>
<!ELEMENT customer-update (customer)>
A sample Schema definition for this XML document type follows:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xsd:schema PUBLIC "-//W3C//DTD XMLSCHEMA 19991216//EN" "" [
        <!ENTITY % p 'xsd:'>
        <!ENTITY % s ':xsd'>
]>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
        <xsd:complexType name="accountType"
 content="empty">
                <xsd:attribute name="account-id"
type="xsd:string" use="required"/>
                <xsd:attribute name="account-name"
```

```
type="xsd:string" use="required"/>
                <xsd:attribute name="account-type"
type="xsd:string" use="required"/>
        </xsd:complexType>
        <xsd:complexType name="customerType"
 content="elementOnly">
                <xsd:sequence>
                        <xsd:element name="account"
type="accountType"/>
                </xsd:sequence>
                <xsd:attribute name="customer-id"
 use="required">
                        <xsd:simpleType base="xsd:binary">
                                <xsd:encoding value="hex"/>
                        </xsd:simpleType>
                </xsd:attribute>
                <xsd:attribute name="first-name"
type="xsd:string" use="required"/>
                <xsd:attribute name="last-name"
type="xsd:string" use="required"/>
                <xsd:attribute name="telephone"
type="xsd:string" use="required"/>
        </xsd:complexType>
        <xsd:element name="customer-update">
                <xsd:complexType content="elementOnly">
                        <xsd:sequence>
                                <xsd:element name="customer"
type="customerType"/>
                        </xsd:sequence>
                        <xsd:attribute name="xmlns:xsi"
type="xsd:uriReference"
use="default"
 value="http://www.w3.org/1999/XMLSchema-instance"/>
                        <xsd:attribute
 name="xsi:noNamespaceSchemaLocation"
type="xsd:string"/>
                        <xsd:attribute
 name="xsi:schemaLocation"
type="xsd:string"/>
                </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

Rewrite all many-to-many relationships by choosing one of the entities of each relationship to be a child element. For example, consider a many-to-many relationship between customer orders and products. This relationship would be rewritten as two one-to-many relationships using the entity `ORDER_LINE_ITEM` as a cross-reference. An `<order-line-item>` element

could be a child of the `<order>` or `<product>` element, or both. Chances are that both do not need to be implemented and that `<order-line-item>` would be considered a child of `<order>`.

Common Mistakes

Declaring attributes as elements. One of the most common XML design mistakes I see is making data elements that should be attributes. For example, some developers would have made `account-name`, from listing 7.2, a separate element instead of an attribute of `<account>`. Misusing elements in this way is likely to cause lower parsing performance and slower XSLT transformations.

Further Reading

Date, C. J. 2003. *An Introduction to Database Systems*, 8th ed. Boston: Pearson/Addison Wesley.

Fleming, Candace C., and Barbara von Halle. 1989. *Handbook of Relational Database Design*. Reading, MA: Addison-Wesley.

8

Network Architecture

The primary goal for network architecture with respect to J2EE applications is to provide the foundation for three important features: security, scalability, and high availability. Although technical architects typically are not responsible for configuring the network, they should understand the features provided by the network architecture to determine what they need to add directly to an application.

The technical architect is responsible for ensuring that applications don't breach the company's security infrastructure. Security in most companies is a centralized function and is treated as a network infrastructure issue. Despite this, the architect must understand what the company infrastructure provides and ensure that any application developed doesn't do anything to circumvent the security architecture in place. For example, with most security architectures I'm aware of, it's technically possible for the application to use a generic ID to provide users access to information for which they're not personally authorized.

The technical architect is responsible for application scalability. Scalability refers to the ability of your site to handle an increasingly large number of users. Although server hardware and network infrastructure provide a platform that makes scaling possible, application design is as much a contributing factor to scalability as the underlying hardware configuration.

The technical architect is responsible for application availability. The term *high availability* describes a site that is always available for use and has minimal downtime. While the server and network infrastructure may provide such features as clustering and automatic fail-over, faulty application design can defeat these features and can make applications unavailable.

Developers who have come up through the ranks as programmers usually have little exposure to networking and network architecture. Because the architecture for J2EE applications typically involves multiple servers, and because I routinely get enough basic networking questions from senior developers, this chapter begins by defining some key networking terms and goes on to explain, by example, the basic functioning of a network.

In addition, the chapter describes and provides examples of two generic architectures commonly used for J2EE applications: one for security and the other for scalability and availability. Understanding your network architecture also helps you identify where a problem is occurring.

Readers who already understand networking basics might want to skip to the section titled "Security."

Key Terms and Concepts

An **IP address** identifies the location of a machine or device on a TCP/IP network. If your PC is on the Internet, you have an IP address. An example IP address is 192.168.1.101. Although it looks as if this address has four parts, it only has two: a network part and a host part. The network part identifies the network where the device is located, and the host part identifies the specific device within that network. In the example, 192.168.1 is the network part, and 101 is the host part. Most networks use the first three nodes of the IP address to identify the network. This is configurable, but that task is beyond the scope of this chapter. Most devices on a network are assigned IP addresses. Network communication between IP addresses is commonly called **traffic**.

A **subnet mask** indicates which parts of the IP address identify the network and which parts identify the host. For instance, a common subnet mask is 255.255.255.0. This means that the first three nodes of an IP address indicates the network, the last node IP address indicates the host. As each node has 256 possible values (0 to 255), this example network can have 256 IP addresses in it.

A **switch** or a **hub** is a device that allows multiple machines to participate on a network. There are technical differences between hubs and switches

that are beyond the scope of this chapter. A switch or hub can also be used to connect networks. For example, my PC is connected to a switch. That switch is connected to another switch that is connected to other machines in my house. Switches and hubs are devices that do not have assigned IP addresses.

A **router** is a device that understands where to send traffic based on the network portion of the IP address. Routers are required for large networks. While hubs and switches allow you to create a network, routers connect entire networks. Some routers are programmable and can actually provide functionality similar to that of firewalls (defined later). Some operating systems are capable of making servers act as routers.

A **gateway** is a router that provides users in a network access to the Internet. Gateways are typically provided by Internet services providers (ISPs).

A **firewall** provides security for a network. The configurable rules of a firewall define what network traffic is allowed and block the rest. Firewalls keep Internet traffic away from servers that corporations use internally. Like many people, I have a firewall in front of my Internet connection to guard against security breaches. Here are a few examples of typical firewall rules:

- ▲ Allow HTTP traffic from anyone to servers ren and stimpy.
- ▲ Allow HTTPS traffic from anyone to servers ren and stimpy.
- ▲ Allow FTP traffic from anyone to server homer.

A **load-balancing appliance** is used for Web sites with high volume. For Web sites that have high numbers of users, sometimes it's cheaper to buy several smaller servers than to buy one or two large ones. The load-balancing appliance distributes traffic over several identically configured Web servers.

A **domain name service (DNS)** tracks labels for IP addresses. For example, it's easier to remember http://www.javasoft.com/, Sun's Java technology Web site, than to remember 192.18.97.39. A DNS tracks the fact that the Java technology Web site is at 192.18.97.39. A DNS also makes it easy for administrators to change the location of a Web site.

A **demilitarized zone (DMZ)** is a network sandwiched between two firewalls. It's common to put a public site in a DMZ, with the Internet outside one firewall and a corporation's internal network outside the other. This decreases the probability of hackers getting into your internal corporate network because they would have to break through two firewalls to do so.

A **cluster** is a group of servers that service the same applications and are configured in such a way that they share one IP address. Clustering is a complex topic and can be defined at a hardware level, software level, or a combination of the two. Sometimes groups of servers like this are referred to as a *server farm*.

Networking Basics

The easiest way to explain how networking works is by example. Let's look at what happens when I surf to http://www.javasoft.com/, the Java technology Web site. My DNS server (63.240.76.4) is asked to provide the IP address for the Java technology Web site, which is 192.18.97.39. My machine sends the request to the gateway/firewall device in my basement (192.168.1.1). That gateway sends the request to my cable modem (IP address unknown). The cable modem sends the request to the gateway provided by my ISP. My communication goes through a large number of routers and gateways illustrated by the traceroute in listing 8.1. Eventually, my request gets to Sun's Java technology Web site.

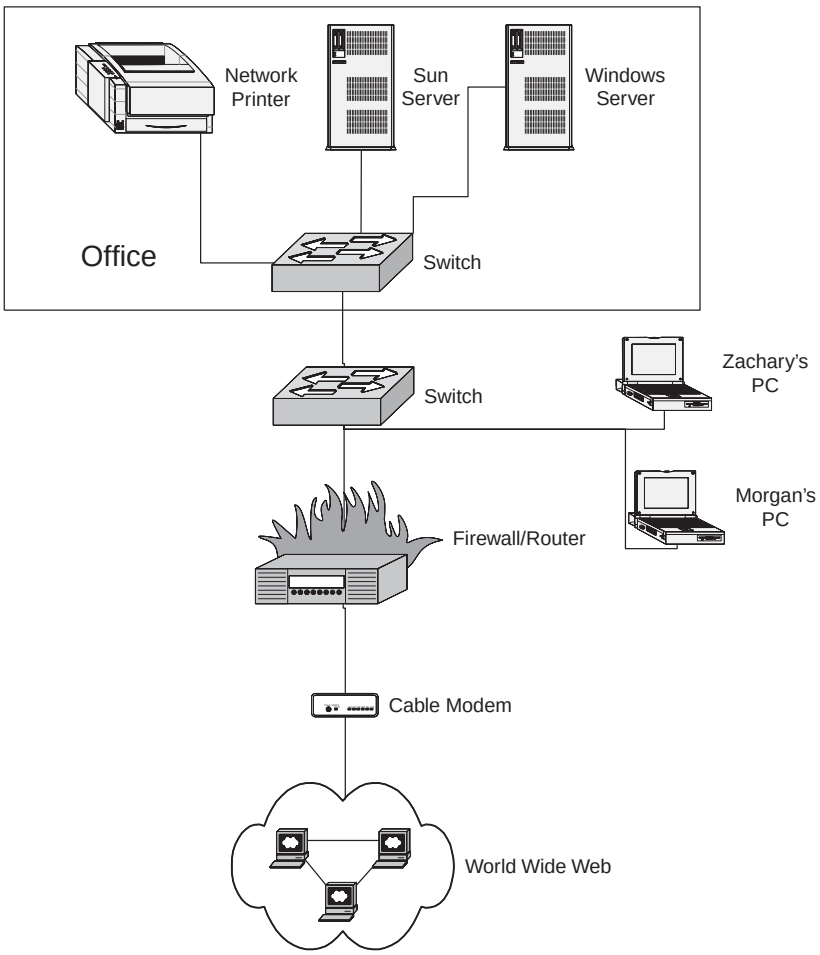
Listing 8.1: A Traceroute to http://www.javasoft.com/

```
C:\>tracert www.javasoft.com

Tracing route to www.javasoft.com [192.18.97.39]
over a maximum of 30 hops:

 1 10 ms 10 ms 10 ms 10.164.160.1
 2 10 ms 10 ms 10 ms 12.244.106.129
 3 10 ms 10 ms 10 ms 12.244.68.26
 4 10 ms 20 ms 10 ms 12.244.68.30
 5 10 ms 10 ms 10 ms 12.244.72.242
 6 10 ms 20 ms 20 ms gbr2-p100.cgcil.ip.att.net [12.123.5.78]
 7 10 ms 20 ms 10 ms tbr1-p013602.cgcil.ip.att.net [12.122.11.37]
 8 10 ms 10 ms 10 ms gbr3-p50.cgcil.ip.att.net [12.123.5.146]
 9 20 ms 10 ms 10 ms POS5-2.BR5.CHI2.ALTER.NET [204.255.169.145]
10 10 ms 10 ms 20 ms 0.so-3-1-0.XL2.CHI2.ALTER.NET [152.63.71.97]
11 20 ms 10 ms 10 ms 0.so-2-0-0.TL2.CHI2.ALTER.NET [152.63.67.109]
12 20 ms 20 ms 10 ms 0.so-7-0-0.TL2.STL3.ALTER.NET [152.63.146.62]
13 70 ms 70 ms 70 ms 0.so-3-0-0.CL2.DEN4.ALTER.NET [152.63.89.233]
14 71 ms 90 ms 80 ms 178.ATM7-0.GW3.DEN4.ALTER.NET [152.63.72.73]
15 * * * Request timed out.
16 80 ms 70 ms 80 ms rwres.java.Sun.COM [192.18.97.39]

Trace complete.
```

Notice that one of the addresses in the route was unidentifiable and was marked "Request timed out" in the trace. This is probably a firewall. The firewall at the Java technology Web site might not be configured to allow traceroutes initiating from the Internet.

Figure 8.1 is an example of a network diagram. The lines indicate network connectivity. All machines on the network must be connected to a hub, switch, or router of some type.

Figure 8.1: Network Diagram Example



Security

Security for J2EE applications and the underlying environment can be thought of in terms of authorization, authentication, and provisioning. Security features are used to limit the types of traffic allowed over a network (e.g., HTTP traffic, mail, FTP, etc.). Further, some transmissions might be allowed for some users but not for all. Firewalls are typically used to limit the types of network transmissions possible at site level. It's standard for J2EE applications to allow only HTTP or HTTPS traffic through the firewall. FTP traffic to an FTP server also might be allowed.

User **authentication** is commonly handled at the Web server level with the assistance of some type of identity management product (such as Oblix™). These products are typically used to force a user login and validate the requested password.

The **authorization** feature of a Web server (with the help of a third-party encryption vendor) is targeted at making sure that the entity providing the user ID and password for the session is still the entity issuing requests over the server. This feature also protects the content of every transmission from being viewed or altered by a third party. SSL is one commonly used encryption mechanism.

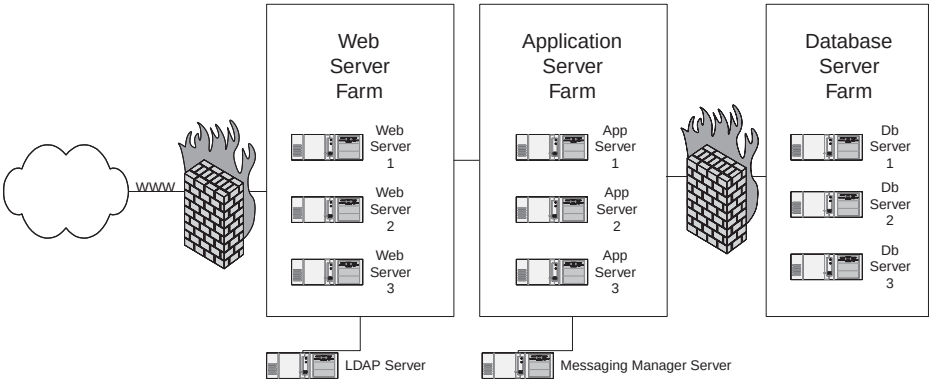
Provisioning features regulate which users get access to which applications. In complex environments, provisioning allows a user to use only some applications, not all. Provisioning is normally provided by a purchased product and not developed internally.

The reason that these concerns are usually addressed at an architecture level is that they're common to all applications. None of the features discussed here should require any specific application-level coding. If the application provides personalization features and the infrastructure provides identity management software, the application might look up information about the user.

Most J2EE applications position their firewalls so that at least the Web and application servers are in a DMZ. Because intruders must break through multiple firewalls to get to application data, using a DMZ reduces the probability of a break-in.

Figure 8.2 depicts a basic network architecture for a J2EE application. In this diagram, the Web servers provide security. The LDAP server is used to provide customer identity information for the Web servers.

Figure 8.2: Basic J2EE Network Architecture



Architecting Application Security

Leverage the infrastructure security features as much as possible. Security features managed at an enterprise level usually have a lower chance of failure compared with custom-coded solutions at the application level. My premise is that commercial and open source packages generally have less severe bugs and are more robust than custom-coded applications.

Audit the use of generic IDs. A generic ID is an ID not associated with a specific person. It is common practice to use a generic ID and password for a database connection. It is not common to use a generic ID to provide protected Web content to users who are not explicitly authorized for that content. I've seen applications that use generic IDs in this way to skirt the security infrastructure of the company.

Effectively use URL masks to indicate security requirements. Most Web servers enforce security based on URLs. For example, a URL containing the phrase "/public" may not be secure, while a URL containing the phrase "/admin" might be restricted to application and system administrators. If the end users of your application have different capabilities depending on who they are, make sure that the URLs used to access them reflect their roles. In most companies, this will increase your chances of being able to use the security infrastructure.

Scalability and High Availability

Scalability and high availability come from the network architecture's provision of redundant Web servers, application servers, and database servers.

Additionally, some architectures might have LDAP and messaging server redundancies.

Hardware redundancies provide scalability by distributing the load over multiple machines. For instance, if your application servers can handle 1,000 concurrent users each, you need five to handle 5,000 users. You can keep adding to your heart's content. The same concept holds for all the other types of servers.

Hardware redundancies provide high availability by reducing the chances that all hardware will be down at the same time and all users prevented from using your site. The more redundancy, the lower the probability of outage (theoretically, at least). Using this logic, diminishing returns appear very quickly. After three servers, the probability of an outage is so low that the probability reductions associated with additional servers are minuscule.

One important corollary to the hardware redundancy principle is that your site is only as strong as your weakest link. One of my clients provides effective redundancies at all levels, except that the company's database doesn't support using multiple servers without manual intervention and some amount of data loss. For this client, adding more servers would achieve little to nothing toward high availability because the company has an unguarded single point of failure anyway. This client's scalability is also limited because the company can only scale the size of the single database server and doesn't have opportunities to increase bandwidth.

The amount of redundancy of each piece of hardware doesn't necessarily increase in parallel. For example, you might add more application servers while leaving the number of Web servers alone. You might increase the number of load balancers and leave the number of Web servers and application servers alone. The roles additional servers play depends on where your CPU cycles are being spent.

System administrators have different ways to bind an army of servers together. One method is to use some type of load-balancing software or appliance. The cheapest tool is a DNS that distributes the load in a round-robin manner. For a large site, however, the company usually purchases a load-balancing appliance that uses a distribution algorithm that is more sophisticated and more efficient than round-robin.

Some operating systems have clustering capabilities, which bind together multiple machines to look like one machine. Because the machines communicate on a low level, if one machine crashes, fail-over is generally quick and the probability of transaction failure during the crash is low.

Operating system clustering is usually complemented by application server software and database software. For example, BEA's WebLogic™ product supports clustering. Oracle's database software also supports clustering (this feature is called Parallel Server).

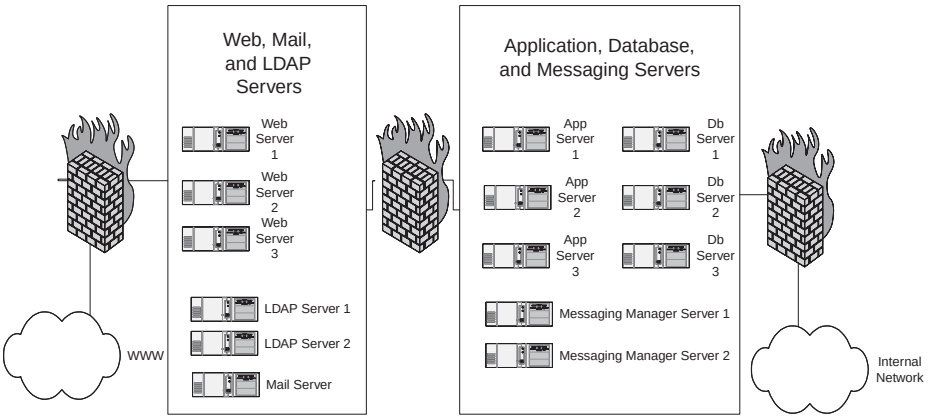
Generally, clustering technologies are costly on a number of fronts. Outright licensing fees are often high for clustered solutions. But far more significant are the support costs associated with effectively administering and configuring such an environment, which you cannot avoid if you want to get the maximum benefit from the software purchase.

Architecting Scalability and Availability

Audit all objects placed on the session to ensure that they are serializable. J2EE applications usually place information on the session, and J2EE containers must be able to serialize these objects and transport them to other servers to use in the event of a failure. If these objects contain anything that doesn't implement `java.io.Serializable`, the container will not be able to provide a seamless transition to another server in response to a failure.

Keep all code servers generic. The clustering capabilities that J2EE containers provide depend on the fact that code executed on one node in the cluster operates the same way when executed in another node. I've seen applications inadvertently make code environment aware and not be able to function in a clustered environment.

Figure 8.3: Network Diagram Example



Network Diagram Example

The architecture shown in figure 8.3 is a slight variation from the generic architecture presented in Figure 8.2. The site has two DMZs for extra security. The outer DMZ hosts Web servers, the LDAP servers (for customer authorization), and a mail server. The inner DMZ hosts all application servers, database servers, and messaging servers. This architecture is redundant and scalable at several levels. As usage of the site expands, the number of Web servers, LDAP servers, application servers, and database servers can easily be expanded to handle growth.

The fact that all types of servers are duplicated at least once (with the exception of the mail server) significantly lowers the probability of an unplanned outage. The fact that the mail server isn't made redundant means that the company is prepared to accept a limited mail service outage.

Further Reading

Hunt, Craig. 2002. *TCP/IP Network Administration*, 3rd ed. Sebastopol, CA: O'Reilly & Associates.

9

Planning Construction

Okay, you've read the chapter title, and you're thinking, "What's a chapter on project planning doing in a technical architect's handbook?" But remember, part of the technical architect's role is to give the project manager information on construction tasks, the order in which they should be completed, and what dependencies exist. And these days, J2EE architects are often called on to fill the project management role.

This chapter develops and adds detail to the high-level requirements described in chapter 3. After completing use-case analysis and object and data modeling, you should have enough information for a more detailed plan. My project plans typically have the following types of activities:

- ▲ Use-case analysis
- ▲ Object modeling
- ▲ Data modeling
- ▲ Data migration/conversion activities
- ▲ Coding and unit testing
- ▲ System testing
- ▲ User acceptance testing
- ▲ Deployment activities

Many of these categories can be divided into lower-level tasks. For example, you could break down use-case analysis into major subject areas of the application. Typically, I divide coding and unit-testing activities into major classes, with one or two people assigned to each.

Figure 9.1 shows a task list from the ProjectTrak application we've been using in the "Architect's Exercise" sections.

Task Order and Dependencies

The most common planning question I get from project managers is how to effectively order construction and unit testing. I usually advise the following order:

- ▲ DAO (with testing classes) and VO classes
- ▲ Business objects (with testing classes)
- ▲ Deployment wrappers and interfaces
- ▲ Presentation layer

Architectural components of a project need to be constructed before they are needed. There is no way to make more detailed recommendations for architectural components because they can be used in all layers of the project.

You will find that most business objects rely heavily on DAOs and VOs. As such, you cannot complete most business objects until the classes they use are complete. If you're using project management software, I'd ensure that these dependencies are properly reflected in the plan.

The presentation layer actions and JSPs would logically be completed after the deployment layer is coded. If the presentation layer coding and construction must start first for political reasons, then stub the deployment wrappers. These "stubs" are throwaway work.

The tasks at this point in the project should become granular enough that most developers will feel comfortable providing estimates. As a result, the project plan can be more accurate now than in the preliminary stages described in chapter 3.

If you're using a project management tool, and the work schedule it computes doesn't make sense, the most likely cause is that some dependencies are incorrect or missing. Many people circumvent the management tool and manually compute and enter start and end dates for all the tasks. I prefer to fix the dependencies rather than produce an unrealistic plan.

Figure 9.1: Example J2EE Project Plan for ProjectTrak
Project Start Date: Thu 6/26/03
Project Finish Date: Wed 11/12/03

Tasks

ID	Task Name	Duration	Start	Finish	Resource Names	% Complete
1	Use case analysis	1 day?	Thu 6/26/03	Thu 6/26/03		0%
2	**Design Activities**	24.5 days	Thu 6/26/03	Wed 7/30/03		0%
3	Screen design and prototype	16 hrs	Mon 7/28/03	Wed 7/30/03	Derek Ashmore	0%
4	**Object Modeling**	4 days	Thu 6/26/03	Tue 7/1/03		0%
5	Base functionality	16 hrs	Thu 6/26/03	Fri 6/27/03	Derek Ashmore	0%
6	Skill set tracking capability	8 hrs	Mon 6/30/03	Mon 6/30/03	Derek Ashmore	0%
7	Baseline capability	8 hrs	Tue 7/1/03	Tue 7/1/03	Derek Ashmore	0%
8	**Data Modeling**	4 days	Tue 7/22/03	Fri 7/25/03		0%
9	Base functionality	16 hrs	Tue 7/22/03	Wed 7/23/03	Derek Ashmore	0%
10	Skill set tracking capability	8 hrs	Thu 7/24/03	Thu 7/24/03	Derek Ashmore	0%
11	Baseline capability	8 hrs	Fri 7/25/03	Fri 7/25/03	Derek Ashmore	0%
12	Physical database design	4 hrs	Mon 7/28/03	Mon 7/28/03	Derek Ashmore	0%
13	**Coding**	46.75 days	Wed 7/2/03	Thu 9/4/03		0%
14	**VO Objects**	30 days	Wed 7/2/03	Tue 8/12/03		0%
15	BaselineVO	2 hrs	Wed 7/9/03	Wed 7/9/03	Developer 1	0%
16	ProjectVO	2 hrs	Wed 7/2/03	Wed 7/2/03	Developer 1	0%
17	ProjectTaskVO	2 hrs	Wed 7/9/03	Wed 7/9/03	Developer 1	0%
18	ProjectTaskWith ProjectionVO	2 hrs	Tue 8/12/03	Tue 8/12/03	Developer 1	0%
19	ResourceVO	2 hrs	Tue 7/8/03	Tue 7/8/03	Developer2	0%
20	SkillsetVO	2 hrs	Wed 7/2/03	Wed 7/2/03	Developer2	0%
21	**Data Access Layer with Test Classes**	15.5 days	Wed 7/2/03	Wed 7/23/03		0%
22	BaselineDAO	40 hrs	Wed 7/16/03	Wed 7/23/03	Developer 1	0%

ID	Task Name	Duration	Start	Finish	Resource Names	% Complete
23	ProjectDAO	40 hrs	Wed 7/2/03	Wed 7/9/03	Developer 1	0%
24	ProjectTaskDAO	40 hrs	Wed 7/9/03	Wed 7/16/03	Developer 1	0%
25	SkillsetDAO	32 hrs	Wed 7/2/03	Tue 7/8/03	Developer2	0%
26	ResourceDAO	32 hrs	Tue 7/8/03	Mon 7/14/03	Developer2	0%
27	**Business Logic Layer with Test Classes**	25.75 days	Wed 7/2/03	Wed 8/6/03		0%
28	ProjectBO	80 hrs	Wed 7/23/03	Wed 8/6/03	Developer 1	0%
29	ResourceBO	40 hrs	Mon 7/14/03	Mon 7/21/03	Developer2	0%
30	TaskScheduler BO	80 hrs	Wed 7/2/03	Tue 7/15/03	Derek Ashmore	0%
31	**Deployment Layer with Client Classes**	16.25 days	Mon 7/21/03	Tue 8/12/03		0%
32	ProjectBean	32 hrs	Wed 8/6/03	Tue 8/12/03	Developer 1	0%
33	ResourceBean	32 hrs	Mon 7/21/03	Fri 7/25/03	Developer2	0%
34	**Presentation Layer**	31.75 days	Wed 7/16/03	Thu 8/28/03		0%
35	Struts configuration	32 hrs	Wed 7/16/03	Mon 7/21/03	Derek Ashmore	0%
36	Test classes with Cactus	40 hrs	Thu 8/21/03	Thu 8/28/03	Derek Ashmore	0%
37	**Action Classes**	14.25 days	Fri 7/25/03	Thu 8/14/03		0%
38	BaselineDisplay Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
39	BaselineSave Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
40	CombinedWork ScheduleDisplay Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
41	ProjectDisplay Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
42	ProjectSave Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
43	TaskDisplay Action	16 hrs	Tue 8/12/03	Thu 8/14/03		0%
44	TaskSaveAction	16 hrs	Tue 8/12/03	Thu 8/14/03		0%

ID	Task Name	Duration	Start	Finish	Resource Names	% Complete
45	WorkSchedule DisplayAction	16 hrs	Fri 7/25/03	Tue 7/29/03		0%
46	WorkSchedule SaveAction	16 hrs	Fri 7/25/03	Tue 7/29/03		0%
47	**JSPs**	5 days	Thu 8/14/03	Thu 8/21/03		0%
48	BaselineDisplay JSP	40 hrs	Thu 8/14/03	Thu 8/21/03		0%
49	ProjectEditDisplay JSP	40 hrs	Thu 8/14/03	Thu 8/21/03		0%
50	ProjectWorkSch eduleDisplayJSP	40 hrs	Thu 8/14/03	Thu 8/21/03		0%
51	TaskEditDisplay JSP	40 hrs	Thu 8/14/03	Thu 8/21/03		0%
52	WorkSchedule EditDisplayJSP	40 hrs	Thu 8/14/03	Thu 8/21/03		0%
53	**Regression Test suite**	5 days	Thu 8/28/03	Thu 9/4/03		0%
54	Code and verify test suite	40 hrs	Thu 8/28/03	Thu 9/4/03	Derek Ashmore	0%
55	System testing	160 hrs	Thu 8/28/03	Thu 9/25/03		0%
56	**User Acceptance Testing**	32 days	Fri 9/26/03	Tue 11/11/03		0%
57	Alpha support	80 hrs	Fri 9/26/03	Fri 10/10/03		0%
58	Beta 1 support	80 hrs	Mon 10/13/03	Mon 10/27/03		0%
59	Beta 2 support	80 hrs	Tue 10/28/03	Tue 11/11/03		0%
60	**Deployment Activities**	99.75 days	Thu 6/26/03	Wed 11/12/03		0%
61	Development environment setup	16 hrs	Thu 6/26/03	Fri 6/27/03		0%
62	System test environment setup	16 hrs	Mon 6/30/03	Tue 7/1/03		0%
63	Alpha release	8 hrs	Thu 9/25/03	Fri 9/26/03		0%
64	Beta 1 release	8 hrs	Fri 10/10/03	Mon 10/13/03		0%
65	Beta 2 release	8 hrs	Mon 10/27/03	Tue 10/28/03		0%
66	Production release	8 hrs	Tue 11/11/03	Wed 11/12/03		0%

Resources

ID	Name	Group	Max Units	Peak Units
1	Technical Architect		100%	100%
2	Developer 1		100%	100%
3	Developer2		100%	100%

Assignments

Task ID	Task Name	Resource Name	Work	Start	Finish	% Work Complete
3	Screen design and prototype	Derek Ashmore	16 hrs	Mon 7/28/03	Wed 7/30/03	0%
5	Base functionality	Derek Ashmore	16 hrs	Thu 6/26/03	Fri 6/27/03	0%
6	Skill set tracking capability	Derek Ashmore	8 hrs	Mon 6/30/03	Mon 6/30/03	0%
7	Baseline capability	Derek Ashmore	8 hrs	Tue 7/1/03	Tue 7/1/03	0%
9	Base functionality	Derek Ashmore	16 hrs	Tue 7/22/03	Wed 7/23/03	0%
10	Skill set tracking capability	Derek Ashmore	8 hrs	Thu 7/24/03	Thu 7/24/03	0%
11	Baseline capability	Derek Ashmore	8 hrs	Fri 7/25/03	Fri 7/25/03	0%
12	Physical database design	Derek Ashmore	4 hrs	Mon 7/28/03	Mon 7/28/03	0%
15	BaselineVO	Developer 1	2 hrs	Wed 7/9/03	Wed 7/9/03	0%
16	ProjectVO	Developer 1	2 hrs	Wed 7/2/03	Wed 7/2/03	0%
17	ProjectTaskVO	Developer 1	2 hrs	Wed 7/9/03	Wed 7/9/03	0%
18	ProjectTaskWith ProjectionVO	Developer 1	2 hrs	Tue 8/12/03	Tue 8/12/03	0%
19	ResourceVO	Developer2	2 hrs	Tue 7/8/03	Tue 7/8/03	0%
20	SkillsetVO	Developer2	2 hrs	Wed 7/2/03	Wed 7/2/03	0%
22	BaselineDAO	Developer 1	40 hrs	Wed 7/16/03	Wed 7/23/03	0%
23	ProjectDAO	Developer 1	40 hrs	Wed 7/2/03	Wed 7/9/03	0%
24	ProjectTaskDAO	Developer 1	40 hrs	Wed 7/9/03	Wed 7/16/03	0%
25	SkillsetDAO	Developer2	32 hrs	Wed 7/2/03	Tue 7/8/03	0%
26	ResourceDAO	Developer2	32 hrs	Tue 7/8/03	Mon 7/14/03	0%
28	ProjectBO	Developer 1	80 hrs	Wed 7/23/03	Wed 8/6/03	0%
29	ResourceBO	Developer2	40 hrs	Mon 7/14/03	Mon 7/21/03	0%

Task ID	Task Name	Resource Name	Work	Start	Finish	% Work Complete
30	TaskScheduler BO	Derek Ashmore	80 hrs	Wed 7/2/03	Tue 7/15/03	0%
32	ProjectBean	Developer 1	32 hrs	Wed 8/6/03	Tue 8/12/03	0%
33	ResourceBean	Developer2	32 hrs	Mon 7/21/03	Fri 7/25/03	0%
35	Struts configuration	Derek Ashmore	32 hrs	Wed 7/16/03	Mon 7/21/03	0%
36	Test classes with Cactus	Derek Ashmore	40 hrs	Thu 8/21/03	Thu 8/28/03	0%
54	Code and verify test suite	Derek Ashmore	40 hrs	Thu 8/28/03	Thu 9/4/03	0%

Critical Path

The **critical path** comprises the dependent tasks that take the longest. In effect, the critical path determines the length of the project. A delay of one day to the critical path will delay the project by one day. Conversely, one day saved in the critical path will allow the project to come in one day early. If you're a technical architect doubling as a project manager, pay more attention to the critical path than to anything else. Most project management software packages highlight the critical path if you have entered all the resource assignments and dependencies completely.

A critical path can shift. If you save enough time on a critical path task, it's possible that it isn't on the critical path anymore—something else is. If a long delay occurs for a task that is not on the critical path but is still essential, the task might become part of the critical path.

For example, financial analysis software I helped develop included a component responsible for generating analysis using company financial information and financial models that users input. In the beginning, this component was not part of the critical path. But as the project proceeded, the critical path changed to incorporate the component because the developer leading the effort to write it didn't have enough knowledge and experience for the task.

The best books I've encountered on the importance of the critical path (and planning in general) are Goldratt (1992, 1997). Although both books use factory assembly lines as examples, the concepts are applicable to J2EE projects (and with their novel-like formats, the books are entertaining reads).

Common Mistakes

Going straight to code. Many developers are impatient with design. They view object-modeling and data-modeling activities as boring compared with coding. I've seen many projects proceed to coding without doing enough modeling to figure out what the target is first. Although most of those projects eventually were finished, they usually used more resources than was necessary. Sometimes, targetless efforts can use two to four times the required resources.

A good analogy is residential construction. When contractors build houses, they create the blueprints first to avoid costly mistakes and rework. Object models and data models are effectively blueprints for J2EE applications.

Permitting a moving target. Once scope is decided for the project (e.g., it has been decided which use cases will be implemented, and the content of those use cases has enough detail from which to design), discourage or even outlaw scope increases. I know this is easier said than done. McConnell (1998) suggests installing a change control board, which is charged with reviewing and authorizing all change requests once a project has progressed passed analysis and high-level design. The existence of a change control board effectively discourages scope increases by creating bureaucratic red tape.

If you can't avoid adding something to a project late in the game, make sure the additional activities, time, and costs get added to the project plan. Also, make sure that the revised project plan reflects the fact that the time spent on analysis and design for the new features zapped time from what you were supposed to be doing (making it late, too). If the new feature causes rework on tasks already completed, make sure that those costs are also documented for all to see.

Think of the residential construction analogy again. Changes in homebuyer wants and desires cause rework and impact the delivery date.

Not correcting personnel assignment mistakes. Of course, it's best to avoid making mistakes in the first place. But when mistakes happen, your best course of action is to recognize and fix them rather than ignore them. The most damaging mistakes in large projects are in the areas of personnel task assignment. This type of mistake is so damaging because most managers are unable to gather the courage to correct the mistakes, thus allowing them to continue.

Although the project manager traditionally handles personnel assignments, the architect (with more knowledge of technical skill sets) should at

least serve in an advisory role. DeMarco and Lister (1999) make some interesting observations:

- The best person on the team outperforms the worst by 10:1.
- The best performer on the team is about 2.5 times better than the average.

In addition, people generally are extremely good at some tasks but poor at others. Good project managers learn to recognize the difference and adjust assignments appropriately. For example, someone might be a whiz when it comes to coding the presentation tier but a complete dud at coding architectural components. Some people can perform testing with ease but are poor at coding.

Saving integration testing activities until the end of the project. Analysis and design mistakes and omissions often aren't visible until construction begins. Integration testing the application makes analysis and design mistakes visible even if the application is only partially functional. Finding these mistakes earlier in the project gives you a chance to correct the error with fewer effects to the project timeline.

Further Reading

DeMarco, Tom, and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams*, 2nd ed. New York: Dorset House.

Goldratt, Eliyahu. 1992. *The Goal: A Process of Ongoing Improvement*. Great Barrington, MA: North River Press.

———. 1997. *Critical Chain*. Great Barrington, MA: North River Press.

McConnell, Steve. 1998. *Software Project Survival Guide*. Redmond, WA: Microsoft Press.

Section 3

Building J2EE Applications

Once the design is complete, the technical architect is often asked to guide application construction. Activities that are the direct responsibility of the technical architect during construction include setting coding standards; mentoring junior developers through more difficult programming tasks; and establishing conventions for logging, exception handling, and application configuration. In addition, the architect (or senior developer) is usually responsible for coding any custom architectural components the application requires because of the difficulty involved in the task.

This section guides you through the application construction process. In it, you will learn how to:

- Establish coding conventions for all software layers.
- Use XML effectively within your application.
- Choose a database persistence method (e.g., JDBC, entity beans, etc.).
- Set conventions and guidelines for transaction management.
- Understand how to make architectural components easy for developers to use.

▲ Set guidelines for logging, exception handling, threading, and configuration management.

This section will also introduce you to the CementJ initiative (http://sourceforge.net/projects/cementj/). CementJ is an open source Java API that provides functionality needed by most Java/J2EE applications that isn't yet provided by the JDK specification directly.

CementJ contains base class value objects, data access objects, enterprise bean clients, application exceptions, and others that can be easily extended and used. CementJ also contains numerous timesaving static utilities that turn common coding tasks into one-liners.

10

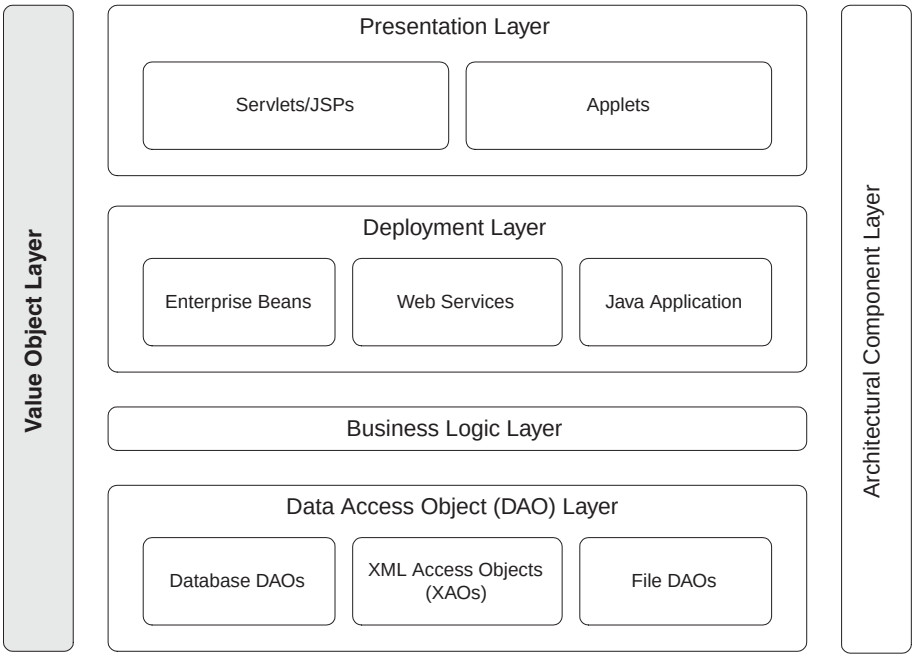
Building Value Objects

A value object (VO) is a lightweight, serializable (i.e., implements `java.io.Serializable`) object that structures groups of data items into a single logical construct. For example, `EmployeeVO` is a class in a human resources application that is a value object containing the data last name, first name, employee ID, job position, and start date. `EmployeeVO` is used in all layers of the application, from presentation to data access. The term *value object* comes directly from the value object design pattern. Some texts use *data transfer object* instead.

The value object design pattern, as written, is intended to minimize network traffic between enterprise beans and their callers (because each argument passed initiates a network transmission). In addition, it is designed to improve the performance of enterprise beans by minimizing the number of method arguments, and thus network transmissions, needed to call them. For example, it's more efficient to call the `addEmployee()` method of an enterprise bean passing an `EmployeeVO` than it is to require individual arguments for last name, first name, and so on. I think of the value object design pattern in much broader terms and use VOs to communicate information between all layers of the application, as illustrated in figure 10.1.

At first glance, my broader definition of a value object appears to contradict the principles of object-oriented design that tell us to combine data

Figure 10.1: Using Value Objects Within a Layered Architecture



with the business logic. Those principles would have us think of "employee" as an object that contains its data (e.g., last name, first name) and methods such as `add()`, `terminate()`, and `oppress()` that represent business logic. For many practical considerations, such as increasing the performance of enterprise beans, we need to have the option for referencing data outside the business logic context.

Chapter 13 will show you ways of constructing objects in the business logic layer that adhere to object-oriented design principles and also allow you to reference the data portion of these objects as a value object. For example, the `Employee` class could easily provide a `getEmployeeVO()` accessor that provides the data for an employee without its business logic.

Because the technical architect is responsible for establishing coding standards and guidelines and mentoring development staff, this chapter provides several implementation tips and techniques for value objects. Additionally, the chapter explains several concepts needed for effectively structuring value objects. And to make implementing these recommendations

easier and less time consuming, the chapter presents a `ValueObject` class, which I've included in the CementJ initiative (http://sourceforge.net/projects/cementj/).

Implementation Tips and Techniques

Always implement `java.io.Serializable`. For a value object to be usable as an argument to any type of distributed object, such as enterprise beans or RMI services, it needs to implement `Serializable`. There are no methods required by `Serializable`, so implementation is easy. You're better off not putting anything in a value object that isn't serializable, such as a database connection. But if you must put a nonserializable object in a value object, declare it `transient` so it's bypassed during any serialization attempts. Listing 10.1 is an extract of value object code.

Listing 10.1: Sample Value Object Code

```
 1:public class CustomerVO
 2:  implements Serializable, Describable
 3:{
 4:  public CustomerVO() {}
 5:
 6:  public String          getCustomerId()
 7:  {
 8: return _customerId;
 9:  }
10:  public void            setCustomerId(String id)
11:  {
12:    if (id == null)
13:    {
14: throw new IllegalArgumentException
15:       ("Null customer Id not allowed.");
16:    }
17:    if (id.equals(""))
18:    {
19: throw new IllegalArgumentException
20:          ("Blank customer Id not allowed.");
21:    }
22:    _customerId = id;
23:  }
24:
25:  // Some code ommitted for brevity
26:  private String          _customerId = null;
27:}
```

Source: /src/book/sample/vo/ CustomerVO.java

Always populate all fields of a value object. Some programmers, for the sake of convenience, don't take the trouble to populate all fields of a value object if they only need a subset of the fields in it. In my experience, this practice saves time during construction, but it inevitably causes bugs that show up as `NullPointerException` exceptions when something attempts to use a field that is not populated. I recommend either populating all fields of a value object or creating a new value object with the new field set.

Always type fields accurately. I've often seen programmers implement dates and numbers as strings, usually to save time when initially coding a value object. But as the application gets larger, this practice can cause confusion and inevitably results in additional conversion code where the field is used. It also leads to confusion for maintenance and causes bugs because someone will format the strings inappropriately.

Check dependence on third-party classes in value objects. Value objects are used as arguments for distributed objects, such as enterprise beans, Web services, and RMI services. If your value objects rely on third-party classes, your callers will have to include them in their classpath to call you. This can be an inconvenience for your callers and make your distributed objects harder to use.

Make value objects self-descriptive. Value objects should have the capability of providing a textual description of themselves for error-logging purposes. One technique that I use to accomplish this is to implement `Describable`.

`Describable` is an interface that specifies how a value object can provide a textual description of itself. Provided with CementJ (package `org.cementj.common`), `Describable` is used in error handling and logging. If you don't provide an easy way to dump the contents of a value object to a log when an exception occurs, it's tedious and time consuming to provide enough detail in the log to be able to reproduce the problem. An example of how tedious exception processing can be without `Describable` is shown in listing 10.2a.

Listing 10.2a: Exception Processing Without Describable

```
1:..... // try block
2:catch (Exception e)
3:{
4:  Logger.logError(
```

```
    "Error updating customer information: " +
    "fname= " + customerVO.getFirstName() +
    "lname= " + customerVO.getLastName() +
    "addr= " + customerVO.getAddress() +
    "city= " + customerVO.getCity() +
    "state= " + customerVO.getState() +
    "zip= " + customerVO.getZipCode())
    ;
}
```

Had `CustomerVO` implemented `Describable`, the exception-handling code would have been much shorter, as illustrated in listing 10.2b.

Listing 10.2b: Exception Processing with Describable

```
..... // try block
catch (Exception e)
{
  Logger.logError (
        "Error updating customer information: " +
        customerVO.describe());
}
```

In some cases, the string value returned by `describe()` could be identical to the results of `toString()`. The difference is that `describe()` results are meant for human eyes. All too often, `toString()` results are programmatically interpreted and not easy for humans to read and interpret. The source code to `Describable` appears in listing 10.3.

Listing 10.3: Describable Interface Definition

```
package org.cementj.common;

public interface Describable
{
  /**
   * Provides a textual version of description and state.
   */
  public String describe();
}
```

Source: src/org/cementj/common/Describable.java

If you prefer not to tie value objects to an outside product such as CementJ, you might consider creating a counterpart to `Describable` in your applications.

Always override method `toString()`. If you don't override `toString()`,

the resulting text is not meaningful. An example from the default implementation of `toString()` is

```
com.myapp.Test@3179c3
```

There are enough classes in the JDK that accept `Object` arguments and expect to be able to `toString()` it that you should provide an implementation (e.g., `StringBuffer`). The implementation of `toString()` inherited from `Object` isn't all that useful.

Consider overriding methods `equals()` and `hashcode()`. If a value object is ever used as a key in a `HashMap`, `Hashtable`, or `HashSet`, `equals()` and `hashcode()` are used for key identification. The definition of these methods, inherited from `Object`, dictates that for two value objects to be equal, they must literally be the same class instance. For example, consider a `CustomerVO` with `firstName` and `lastName` fields. You could have two instances of "John Doe" that will look unequal using the `equals()` inherited from `Object`. You will have to override both `equals()` and `hashcode()` for a value object if you want it usable in any type or `Map` object, such as `Hashtable`, `HashMap`, or `TreeMap`.

The behavior differences between a meaningful implementation of `equals()` and the implementation inherited from `Object` confuses many developers. The example in Listing 10.4a should help alleviate any confusion.

Listing 10.4a: Sample Object.equals() Implementation

```
 1:    public void showObjectEqualImplementation()
 2:    {
 3:      ObjectWithoutEqualsImpl fiveAsObject =
 4:            new ObjectWithoutEqualsImpl("5");
 5:      ObjectWithoutEqualsImpl anotherFiveAsObject =
 6:            new ObjectWithoutEqualsImpl("5");
 7:      ObjectWithoutEqualsImpl sevenAsObject =
 8:            new ObjectWithoutEqualsImpl("7");
 9:
10:      System.out.println("Object equals() demo:");
11:      System.out.println(
12:        "\tfiveAsObject.equals(anotherFiveAsObject): "+
13:      fiveAsObject.equals(anotherFiveAsObject));
14:      System.out.println(
15:         "\tfiveAsObject.equals(sevenAsObject): " +
16:        fiveAsObject.equals(sevenAsObject));
17:    }
```

Source: src/book/sample/general/EqualsDemonstration.java

Listing 10.4a uses a simple class that does not override method `equals()` and uses the implementation inherited from `Object`. The variable declared in line 3 with the value 5 should be "equal" to the object declared in line 5. However, if you were to run the sample, you would see that the variables are actually not considered equal. Output to the sample is provided in listing 10.4b.

Listing 10.4b: Output from Listing 10.4a

```
Object equals() demo:
        fiveAsObject.equals(anotherFiveAsObject): false
        fiveAsObject.equals(sevenAsObject): false
```

If you were to run a different sample using class `String` instead of `ObjectWithoutEqualsImpl`, the output would be more what you would expect because `String` overrides method `equals()`.

The method `hashcode()` returns an integer that is guaranteed to be equal for two instances of `Object` that are equal. The logic behind constructing an algorithm to do this can get intricate. I usually utilize `String`, which has a nice implementation of `hashcode()`. You can concatenate all fields of a value object and get the hashcode of the resulting string. It is legal to have `hashcode()` return the same integer for all instances, but this will make using `HashMap` and `Hashtable` extremely inefficient. Listing 10.5 illustrates an effective implementation of `hashcode()`.

Listing 10.5: Sample hashcode() Implementation

```
 1:  public int hashcode()
 2:  {
 3:    return this.getObjectAsString().hashCode();
 4:  }
 5:
 6:  private String getObjectAsString()
 7:  {
 8:    return this.getObjectAsString(this);
 9:  }
10:
11:  private String getObjectAsString(CustomerVO vo)
12:  {
13:    StringBuffer buffer = new StringBuffer(256);
14:
15:    if (vo._customerId != null)
16:    {
17:  buffer.append(vo._customerId);
```

```
18:    }
19:    else buffer.append("null");
20:    if (vo._firstName != null)
21:    {
22:  buffer.append(vo._firstName);
23:    }
24:    else buffer.append("null");
25:    if (vo._lastName != null)
26:    {
27:  buffer.append(vo._lastName);
28:    }
29:    else buffer.append("null");
30:    if (vo._address != null)
31:    {
32:  buffer.append(vo._address);
33:    }
34:    else buffer.append("null");
35:    if (vo._city != null) buffer.append(vo._city);
36:    else buffer.append("null");
37:    if (vo._state != null) buffer.append(vo._state);
38:    else buffer.append("null");
39:    if (vo._zipCode != null)
40:    {
41:  buffer.append(vo._zipCode);
42:    }
43:    else buffer.append("null");
44:
45:    return buffer.toString();
46:  }
```

Source: /src/book/sample/vo/ CustomerVO.java

Implementing `equals()` is similar. You concatenate all field members and use the `equals()` implementation of `String`, as shown in listing 10.6.

Listing 10.6: Sample equals() Implementation

```
1:  public boolean equals(Object obj)
2:  {
3:  boolean answer = false;
4:
5:  if (obj instanceof CustomerVO)
6:  {
7:  String dtoId =
8:    this.getObjectAsString( (CustomerVO) obj );
9:  if (this.getObjectAsString().equals(dtoId))
10: {
11:   answer = true;
12: }
```

```
13: }
14:
15: return answer;
16: }
```

Source: /src/book/sample/vo/ CustomerVO.java

Consider implementing `java.lang.Comparable`. If you ever use a value object in a sorted collection (e.g., `TreeSet` or `TreeMap`), you must implement `Comparable` for sensible sort results. Implementing `Comparable` requires the implementation of a `compareTo()` method that returns 0 if the two objects are equal, a negative number if the object is less than the argument passed, or a positive number if the `Object` is greater than the argument passed. Listing 10.7 illustrates.

Listing 10.7: Sample compareTo() Implementation

```
 1:  public int compareTo(Object obj)
 2:  {
 3:    int compareResult = 0;
 4:    Object tempObj = null;
 5:    Object tempObjCompareTarget = null;
 6:    Comparable c1, c2;
 7:
 8:    if (obj == null)
 9:    {
10: throw new IllegalArgumentException
11: ("Comparisons to null objects not defined.");
12:    }
13:    if (! (obj instanceof CustomerVO) )
14:    {
15: throw new IllegalArgumentException
16:     ("Comparing different class types not allowed.");
17:    }
18:
19:    CustomerVO dto = (CustomerVO) obj;
20:    compareResult = _lastName.compareTo(dto._lastName);
21:    if (compareResult == 0)
22:    {
23: compareResult =
24:         _firstName.compareTo(dto._firstName);
25:    }
26:    if (compareResult == 0)
27:    {
28: compareResult =
29:         _customerId.compareTo(dto._customerId);
30:    }
31:    if (compareResult == 0)
```

```
32:    {
33:  compareResult =
34:          _address.compareTo(dto._address);
35:    }
36:    if (compareResult == 0)
37:    {
38:  compareResult =
39:          _city.compareTo(dto._city);
40:    }
41:    if (compareResult == 0)
42:    {
43:  compareResult = _state.compareTo(dto._state);
44:    }
45:    if (compareResult == 0)
46:    {
47:  compareResult =
48:          _zipCode.compareTo(dto._zipCode);
49:    }
50:
51:    return compareResult;
52:  }
```

Source: /src/book/sample/vo/ CustomerVO.java

Value Objects Made Easy

My laundry list of recommendations makes implementing value objects extremely boring, tedious, and time consuming. As a technical architect, you have the option of mentoring developers as they follow these recommendations and, you hope, implement them consistently. Another option is to provide architectural utilities that make coding value objects easier and quicker and bring some consistency to value object behavior.

These goals are achieved by CementJ, a tool I created to provide architectural support for value objects. To use CementJ value object support, you need to extend `org.cementj.base.ValueObject` from the CementJ API. `ValueObject` contains a meaningful implementation of `equals()`, `hashcode()`, and `describe()` so you don't have to implement them. Unfortunately, CementJ isn't able to provide a reliable implementation of `compareTo()` because of current limitations with Java's reflection API. Listing 10.8 is an example of how you can extend `ValueObject`.

Listing 10.8: Sample Value Object Extending org.cementj.base.ValueObject

```
 1:package book.sample.vo.cementj;
 2:
 3:import org.cementj.base.ValueObject;
 4:
 5:public class CustomerVO extends ValueObject
 6:{
 7:
 8:  public CustomerVO() {}
 9:
10:  public String          getCustomerId()
11:  {
12: return _customerId;
13:  }
14:  public void            setCustomerId(String id)
15:  {
16:    if (id == null)
17:    {
18: throw new IllegalArgumentException
19:                ("Null customer Id not allowed.");
20:    }
21:    if (id.equals(""))
22:    {
23: throw new IllegalArgumentException
24:                ("Blank customer Id not allowed.");
25:    }
26:    _customerId = id;
27:  }
28:
29:  // some code ommitted for bevity
30:
31:  private String          _customerId = null;
32:  private String          _firstName = null;
33:  private String          _lastName = null;
34:  private String          _address = null;
35:  private String          _city = null;
36:  private String          _state = null;
37:  private String          _zipCode = null;
38:}
```

Source: /src/book/sample/vo/cementj/CustomerVO.java

Although listing 10.8 has the same functionality as the custom-coded original value object, it's only 73 lines long as opposed to 180. You don't need to specify implementation of `Serializable` or `Describable` because `ValueObject` does it for you.

In addition, `ValueObject` provides a `describeAsXML()` feature that formats the content of a value object into an XML formatted document. This feature can save you time in applications that need to transmit the content of a value object via messaging technologies to another application. Another possible use of this feature is to use the XML version of the content to provide better presentation in error messages. Listing 10.9 illustrates an XML document generated by `ValueObject`.

Listing 10.9: Sample XML Document Description of a ValueObject

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book-sample-dto-cementj-CustomerDTO>
<book-sample-dto-cementj-CustomerDTO name="book-sample-dto-cementj-
CustomerDTO">
   <Field name="_customerId" type="java.lang.String"
          value="12345"/>
   <Field name="_firstName" type="java.lang.String"
          value="Derek"/>
   <Field name="_lastName" type="java.lang.String"
          value="Ashmore"/>
   <Field name="_address" type="java.lang.String"
          value="34 Yorktown Center, PMB 400"/>
   <Field name="_city" type="java.lang.String"
          value="Lombard"/>
   <Field name="_state" type="java.lang.String"
          value="IL"/>
   <Field name="_zipCode" type="java.lang.String"
          value="60148"/>
</book-sample-dto-cementj-CustomerDTO>
```

`ValueObject` does use reflection to achieve its magic, so it's slower than custom-coded value objects. Table 10.1 presents a performance comparison for our sample `CustomerVO`.

Table 10.1: Performance Comparison for ValueObject for 1,000

Version	JVM	Milliseconds per 1,000 operations		
		Instantiation	`HashCode()`	`Equals()`
VO extension	1.4.2	241	291	270
Custom	1.4.2	0	0	40
VO extension	1.3.1	290	40	270
Custom	1.3.1	0	0	90

For the table, I arbitrarily chose 1,000 iterations to make the time differences more apparent. Most applications will have fewer executions of these methods per transaction.

By using `ValueObject`, you trade some performance for development and maintenance time. Most value objects in an application use these operations in large enough volume for the speed of `ValueObject` to be an issue. I recommend you custom code only the small number of value objects that you've determined need faster execution time based on your performance-tuning efforts.

In addition, since value objects that extend `ValueObject` contain only small amounts of logic, if any, no test cases are necessary. If you choose to provide your own implementations of `equals()`, `hashcode()`, and `toString()`, you need to construct test cases for these methods.

Common Mistakes

Populating VOs inconsistently. I've seen some applications where fields of value objects were populated with different value sets depending on usage context. This practice is error prone and leads to bugs. It's also confusing for developers to maintain, especially if they weren't involved in the initial development effort. Usually, this practice is a red flag indicating that the design is process oriented instead of object oriented.

Using a blank string to avoid a `NullPointerException`. Some developers initialize all fields to a blank string or something that means "null" but really isn't, as in the following:

```
private String  _customerName = "";
```

Although this kind of declaration eliminates `NullPointerException`

`exceptions`, it doesn't prevent some other type of derivative exception from appearing down the line. This practice is akin to "sweeping dirt under the rug" and is best avoided.

Maintaining parent-child relationships for VOs in both directions. For example, the `CustomerVO` would contain a collection of `AccountVO` children, and each of the `AccountVO` instances would contain a reference back to the parent. You run into the same problems with this practice as you do when you replicate data in relational databases. The result is double the maintenance when data is changed. Further, the practice is error prone and tends to be the root cause of bugs.

Architect's Exercise: ProjectTrak

For the ProjectTrak application, which we've been working on in the "Architect's Exercises" throughout this book, we will use CementJ and its `ValueObject` for all value objects. Listing 10.10 has code from the `ProjectVO` class of ProjectTrak. You may recall that we identified this object in chapter 6.

In addition to implementing all the coding recommendations presented in this chapter, I suggest you check the arguments to mutators to catch illegal variable assignments early. This practice reduces the likelihood that you'll get a derivative `NullPointerException` of something else that's relatively time consuming to debug.

Listing 10.10: Sample ProjectVO Class from ProjectTrak

```
 1:package com.dvt.app.project.vo;
 2:
 3:import org.cementj.base.ValueObject;
 4:import java.util.Date;
 5:import java.io.Serializable;
 6:
 7:/**
 8: *  VO representing information about a project.  This
 9: *  class is part of the ProjectTrak application.
10: *  <p>Copyright:  Delta Vortex Technologies, 2003.
11: */
12:public class ProjectVO
13:    extends ValueObject
14:    implements Serializable, Comparable
15:{
16:
17:  public ProjectVO() {}
```

```
18:
19:  public String getProjectName()
20:  {
21:      return _projectName;
22:  }
23:  public void setProjectName(String name)
24:  {
25:    if (name == null)
26:    {
27:        throw new IllegalArgumentException
28:                ("Null project name not allowed.");
29:    }
30:    if (name.equals(""))
31:    {
32:        throw new IllegalArgumentException
33:                ("Blank project name not allowed.");
34:    }
35:    _projectName = name;
36:  }
37:
38:  public String[] getProjectBaselineNames()
39:  {
40:      return _projectBaselines;
41:  }
42:  public void setProjectBaselineNames(String[] name)
43:  {
44:    _projectBaselines = name;
45:  }
46:
47:  public Date getDateCreated()
48:  {
49:      return _dateCreated;
50:  }
51:  public void setDateCreated(Date dateCreated)
52:  {
53:    if (dateCreated == null)
54:    {
55:        throw new IllegalArgumentException
56:                ("Null dateCreated not allowed.");
57:    }
58:    _dateCreated = dateCreated;
59:  }
60:
61:  public Date getDateModified()
62:  {
63:      return _dateLastModified;
64:  }
65:  public void setDateModified(Date dateLastModified)
66:  {
```

```
67:    if (dateLastModified == null)
68:    {
69:        throw new IllegalArgumentException
70:                ("Null dateLastModified not allowed.");
71:    }
72:    _dateLastModified = dateLastModified;
73:  }
74:
75:  public Date getProjectStart()
76:  {
77:      return _projectStart;
78:  }
79:  public void setProjectStart(Date start)
80:  {
81:    _projectStart = start;
82:  }
83:
84:  public Date getProjectEnd()
85:  {
86:      return _projectEnd;
87:  }
88:  public void setProjectEnd(Date end)
89:  {
90:    _projectEnd = end;
91:  }
92:
93:  public ResourceVO[] getAssignedResources()
94:  {
95:      return _assignedResources;
96:  }
97:  public void setAssignedResources(
98:                       ResourceVO[] assignedResources)
99:  {
100:   if (assignedResources == null)
101:   {
102:       throw new IllegalArgumentException
103:             ("Null assignedResources not allowed.");
104:   }
105:   _assignedResources = assignedResources;
106: }
107:
108: public ProjectTaskVO[] getProjectTasks()
109: {
110:     return _projectTasks;
111: }
112: public void setProjectTasks(
113:                          ProjectTaskVO[] projectTasks)
114: {
115:   if (projectTasks == null)
```

```
116:    {
117:        throw new IllegalArgumentException
118:                ("Null projectTasks not allowed.");
119:    }
120:    _projectTasks = projectTasks;
121:  }
122:
123:  public int compareTo(Object obj)
124:  {
125:      int comparator = 0;
126:       if (obj == null)
127:       {
128:           throw new IllegalArgumentException
129:               ( "Null object not allowed.");
130:       }
131:     if (! (obj instanceof ProjectVO) )
132:     {
133:         throw new IllegalArgumentException
134:             ( "Invalid Object Type: " +
135:               obj.getClass().getName());
136:     }
137:
138:     ProjectVO pvo = (ProjectVO) obj;
139:     comparator = _projectName.compareTo(
140:                         pvo._projectName);
141:     if (comparator == 0)
142:     {
143:        comparator = _dateLastModified.compareTo(
144:                             pvo._dateLastModified);
145:     }
146:
147:     return comparator;
148:  }
149:
150:  private String            _projectName = null;
151:  private String[]          _projectBaselines = null;
152:  private Date              _dateCreated = null;
153:  private Date              _dateLastModified = null;
154:  private ResourceVO[]      _assignedResources = null;
155:  private ProjectTaskVO[]   _projectTasks = null;
156:  private Date              _projectStart = null;
157:  private Date              _projectEnd = null;
158:
159:}
```

Source: /src/com/dvt/app/project/vo/ProjectVO.java

11

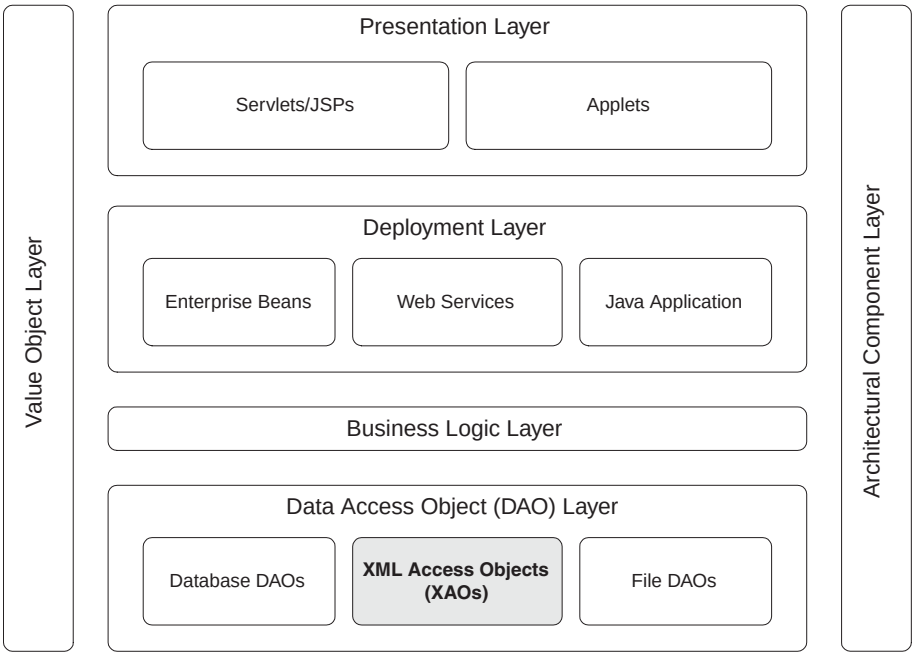
Building XML Access Objects

Data access objects are classes that read and write persistent data. XML manipulation, because it's really a data access operation, is part of the DAO layer. An XML access object (XAO) reads and writes data in an XML format and converts that format to value objects that other layers in the application can use. For example, `PurchaseOrderXAO` is an XAO for a purchasing application that reads and transmits orders in an XML format. `PurchaseOrderXAO` contains the following methods:

```
public void setPurchaseOrder(String xmlText);
public void setPurchaseOrder(PurchaseOrderVO[] order);
public void setPurchaseOrder(InputStream order);

public String        getPurchaseOrderXmlText();
public PurchaseOrderVO[]  getPurchaseOrder();

public void transmit(EDIDestination dest);
public void save(OutputStream os);
```

Business objects use XAOs to interpret and produce XML data, as illustrated in figure 11.1. Typically, XAOs should have little to do with implementing the business rules associated with processing the data. XML-related code is separated to limit and localize the impact that changes in the XML structure have on your application. If this logic were scattered in various

Figure 11.1: Using XML Access Objects Within a Layered Architecture



places in the business logic layer, for example, it would be much harder to find and change.

As a technical architect, you are responsible for forming coding standards and guidelines. This chapter provides implementation guidance and examples for structuring XAOs. In addition, we'll look at a way to generate code that XAOs can use easily, saving you development and maintenance time.

The chapter assumes that you have a basic knowledge of XML concepts and have used an XML parser with Java. Familiarity with XSL style sheets and templates will also help you understand the examples presented here. For readers wanting an XML concept review, the tutorials at W3Schools (http://www.w3schools.com/) are well written and concise.

An XAO Example

XML access objects are responsible for translating XML documents into value objects that can be used by the rest of the application and vice versa.

For example, Listing 11.1 illustrates the setPurchaseOrder() method that reads an XML document and extracts an array of purchase order value objects. This example happens to use the JAXB API along with a CementJ utility to interpret XML. However, JDOM fans or developers who like the native DOM parser would place their extraction logic here. In fact, your XML parsing and interpretation strategy can change in XAOs without adversely affecting the rest of your application.

Listing 11.1: Sample XAO Method to Read an XML Document

```
 1:package book.sample.dao.xml;
 2:
 3:import org.cementj.util.JAXBUtility;
 4:// some imports omitted.
 5:
 6:public class PurchaseOrderXAO
 7:{
 8:  private static final String
 9:      PURCHASE_ORDER_JAXB_PACKAGE =
10:        "book.sample.dao.xml.po";
11:
12:  public void setPurchaseOrder(InputStream xmlStream)
13:  {
14:    PurchaseOrderVO[] poArray = null;
15:    ArrayList poList = new ArrayList();
16:    PurchaseOrderVO po = null;
17:    CustomerOrderType xmlOrder = null;
18:
19:    try
20:    {
21:      CustomerOrderList order =
22:        (CustomerOrderList)
23:            JAXBUtility.getJaxbXmlObject(
24:                 PURCHASE_ORDER_JAXB_PACKAGE,
25:                xmlStream );
26:      List xmlOrderList = order.getCustomerOrder();
27:      for (int i = 0 ; i < xmlOrderList.size(); i++)
28:      {
29:        xmlOrder = (CustomerOrderType)
30:                        xmlOrderList.get(i);
31:        po = new PurchaseOrderVO();
32:
33:        po.setCustomerId(xmlOrder.getCustomerId());
34:        po.setOrderNbr(
35:            Integer.parseInt( xmlOrder.getOrderId() ));
36:        // ... Other Purchase Order information
37:        //     gathered here.
```

```
38:
39:         poList.add(po);
40:       }
41:
42:       if (poList.size() > 0)
43:       {
44:         poArray = new PurchaseOrderVO[poList.size()];
45:         poArray = (PurchaseOrderVO[])
46:                     poList.toArray(poArray);
47:       }
48:
49:       this.setPurchaseOrder(poArray);
50:     }
51:     catch (Throwable t)
52:     {
53:       throw new SampleException(
54:         "Error parsing PO XML.", t);
55:     }
56:   }
57:}
```

Source: /src/book/sample/dao/xml/PurchaseOrderXAO.java

I used a utility from CementJ in line 23 to save several lines of code. Line 23 could easily be replaced with code using JAXB or JDOM directly if you prefer.

A method to create XML documents would be structured much the same way. The `PurchaseOrderXAO` class could easily have a method called `getPurchaseOrderXmlText()` that generates XML text, as illustrated in listing 11.2.

Listing 11.2: Sample XAO Method to Create XML Text

```
 1:package book.sample.dao.xml;
 2:
 3:import org.cementj.util.JAXBUtility;
 4:// some imports omitted.
 5:
 6:public class PurchaseOrderXAO
 7:{
 8:  private static final String
 9:      PURCHASE_ORDER_JAXB_PACKAGE =
10:        "book.sample.dao.xml.po";
11:
12:  public String getPurchaseOrderXmlText()
13:  {
14:    String xmlText = null;
15:    ObjectFactory factory = new ObjectFactory();
```

```
16:
17:     CustomerOrderType xmlOrder = null;
18:
19:     try
20:     {
21:       CustomerOrderList xmlOrderList =
22:           factory.createCustomerOrderList();
23:       for (int i = 0; i < _purchaseOrder.length; i++)
24:       {
25:         xmlOrder = factory.createCustomerOrderType();
26:         xmlOrder.setCustomerId(
27:             _purchaseOrder[i].getCustomerId());
28:         xmlOrder.setOrderId(
29:             Integer.toString(
30:                 _purchaseOrder[i].getOrderNbr() ));
31:         // ... Other Purchase Order information set
32:         //     here.
33:
34:         xmlOrderList.getCustomerOrder().add(xmlOrder);
35:       }
36:
37:       xmlText = JAXBUtility.flushXmlToString(
38:             PURCHASE_ORDER_JAXB_PACKAGE, xmlOrderList);
39:     }
40:     catch (JAXBException j)
41:     {
42:       throw new SampleException(
43:         "Error creating PO XML.", j);
44:     }
45:
46:     return xmlText;
47:   }
48:}
```

Source: /src/book/sample/dao/xml/PurchaseOrderXAO.java

Notice that XML interpretation and generation are self-contained and localized. If attributes are added to the `<purchase-order>` XML document and are needed by the application, those changes are localized to XAO classes and those generated by JAXB. The XML document format can change without affecting the other layers of the application.

Sometimes, XAOs are used to translate XML documents into alternative formats. A common technology to accomplish this is XSLT. As an example, I've created a short XSL template that translates the `<purchase-order>` XML document into HTML, which can be sent to a browser. The `PurchaseOrderXAO` class could easily have a method called

`getPurchaseOrderAsHtml()` that generates XML text, as illustrated in listing11.3.

Listing 11.3: Sample XAO Method to Create HTML Text

```
 1:package book.sample.dao.xml;
 2:
 3:import javax.xml.transform.Transformer;
 4:import javax.xml.transform.TransformerFactory;
 5:import javax.xml.transform.stream.StreamResult;
 6:import javax.xml.transform.stream.StreamSource;
 7:// some imports omitted.
 8:
 9:public class PurchaseOrderXAO
10:{
11:
12:  public String getPurchaseOrderAsHtml()
13:  {
14:      String htmlText = null;
15:      String xmlText = this.getPurchaseOrderXmlText();
16:    ByteArrayInputStream xmlTextStream =
17:        new ByteArrayInputStream(xmlText.getBytes());
18:
19:      try
20:      {
21:        ByteArrayOutputStream output =
22:                new ByteArrayOutputStream
23:                    (xmlText.length() * 2);
24:        TransformerFactory tFactory =
25:                TransformerFactory.newInstance();
26:        Transformer transformer =
27:            tFactory.newTransformer
28:               (new StreamSource("PurchaseOrder.xsl"));
29:            transformer.transform(
30:                new StreamSource(xmlTextStream),
31:                       new StreamResult(output) );
32:        htmlText = output.toString();
33:    }
34:    catch (Throwable t)
35:    {
36:        throw new SampleException(
37:          "Error creating PO HTML.", t);
38:    }
39:
40:      return htmlText;
41:  }
42:}
```

Source: /src/book/sample/dao/xml/PurchaseOrderXAO.java

Architectural Guidelines

Avoid direct use of the DOM to interpret XML data. It is faster to develop and easier to maintain applications using complementary technologies, such as JAXB or JDOM, than using a DOM parser directly.

My preference is JAXB, which generates Java source code that can read, interpret, and serialize XML documents conforming to a given schema. The advantages of JAXB are that it maps XML documents to Java classes that are easy for developers to use, and the learning curve for JAXB is short.

Apply XML technologies consistently throughout the application. Whatever your technology choices are, there is a tremendous benefit to consistency. For instance, if the developers of your application prefer JDOM and are comfortable with that choice, you have little reason to use JAXB. Consistency makes the application easier to maintain because it reduces the skill sets required for maintenance developers. Consistency also reduces the time it takes to investigate bugs because maintenance developers can begin with a base understanding as to how XAOs are structured.

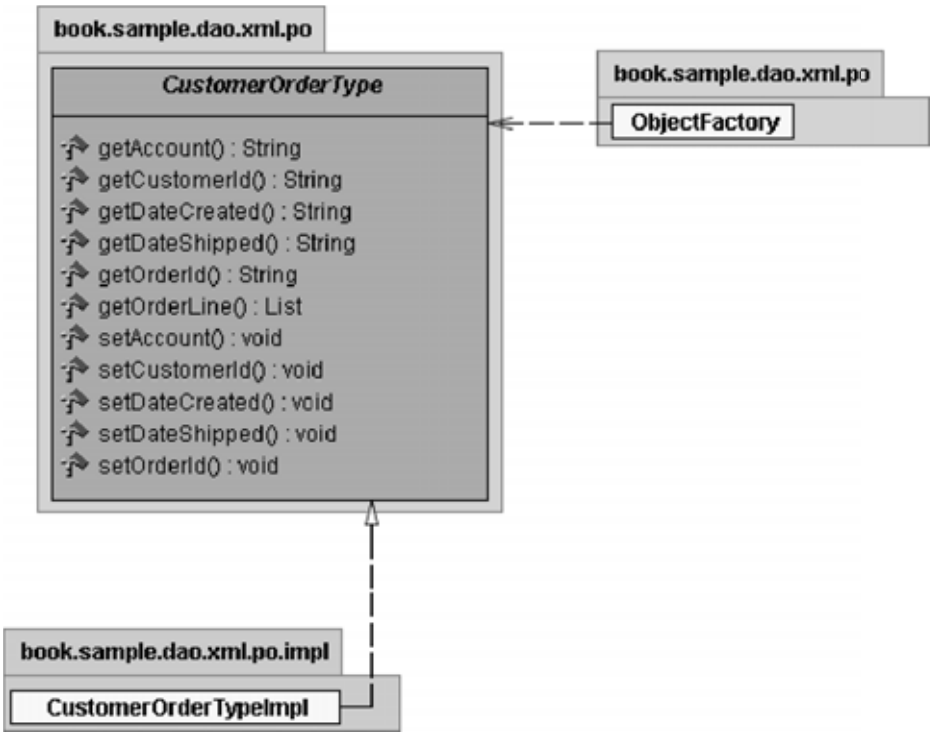
Place XML-related code into separate classes. One reason for separating XML-related classes from those that implement business rules is to insulate your application from changes in XML document structure. Another reason is that separating XML document interpretation and business logic can lead to simpler code. Further, if multiple applications must read and interpret the same XML document formats, keeping XML-related code separate makes it easier to share that code across applications.

Overview of JAXB

Because JAXB is a relatively new API, here is a brief synopsis of JAXB and an example of how to use it in this section. JAXB can be downloaded from the Java Web site (http://java.sun.com/xml/jaxb/). Refer to the JAXB documentation for installation information.

JAXB is a code generator that provides Java classes that mimic an XML document format. Each XML element is generated into a Java class with fields for all attributes on the XML document. For example, a <customer-order> XML element might be generated into a `CustomerOrderType` interface that has accessors and mutators for all attributes and child elements of the segment. Programmers can then use the generated class as they would

Figure 11.2: Object Model for JAXB-Generated Interface



any value object they would write. A diagram of the `CustomerOrderType` interface generated by JAXB is presented in Figure 11.2.

A similar interface and implementation was generated by JAXB for all elements in the schema. An important thing to note is that child elements are represented as lists. These generic lists contain only JAXB-generated components.

Instantiating a JAXB object from a file containing an XML document is relatively easy. Listing 11.4 is an example.

Listing 11.4: Using JAXB to Read an XML Document

```
1:import javax.xml.bind.JAXBContext;
2:import javax.xml.bind.Marshaller;
3:import javax.xml.bind.Unmarshaller;
4:
5:public class SampleJAXB
6:{
```

```
 7:  public Object getJAXBObject() throws Throwable
 8:  {
 9:    InputStream xmlDocumentInputStream =
10:        new FileInputStream( "PurchaseOrder.xml" );
11:    JAXBContext jc = JAXBContext.newInstance(
12:        "book.sample.dao.xml.po" );
13:    Unmarshaller u = jc.createUnmarshaller();
14:    return u.unmarshal( xmlDocumentInputStream );
15:  }
16:}
```

Source: /src/book/sample/dao/xml/SampleJAXB.java

The JAXB calls are virtually identical for all JAXB documents. Alternatively, you can use the CementJ `JAXBUtility` (from package `org.cementj.util`) as a shortcut, as shown in listing 11.5.

Listing 11.5: Using CementJ JAXBUtility to Read an XML Document

```
1:import org.cementj.util.JAXBUtility;
2:
3:public class SampleJAXB
4:{
5:  public Object getJAXBObjectViaCementJ()
6:      throws Throwable
7:  {
8:    return JAXBUtility.getJaxbXmlObject(
9:        "book.sample.dao.xml.po",
10:       new File("PurchaseOrder.xml") );
11:  }
12:}
```

Source: /src/book/sample/dao/xml/SampleJAXB.java

Similarly, producing an XML document from a JAXB-binded object is also relatively easy. Listing 11.6 provides an example.

Listing 11.6: Using JAXB to Write an XML Document to Disk

```
1:import javax.xml.bind.JAXBContext;
2:import javax.xml.bind.Marshaller;
3:import javax.xml.bind.Unmarshaller;
4:
5:public class SampleJAXB
6:{
7:  public void serializeJAXBObject(
8:          CustomerOrderList order)
9:      throws Throwable
10:  {
```

```
11:    JAXBContext jc = JAXBContext.newInstance(
12:        "book.sample.dao.xml.po" );
13:    Marshaller m = jc.createMarshaller();
14:    m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT,
15:        Boolean.TRUE );
16:    m.marshal( order, new FileOutputStream(
17:        "PurchaseOrderGenerated.xml") );
18:  }
19:}
```

Source: /src/book/sample/dao/xml/SampleJAXB.java

CementJ also provides a shortcut for writing XML documents, as listing 11.7 illustrates.

Listing 11.7: Using CementJ to Produce an XML Document with JAXB Classes

```
1:import org.cementj.util.JAXBUtility;
2:
3:public class SampleJAXB
4:{
5:  public void serializeJAXBObjectWithCementJ(
6:        CustomerOrderList order)
7:    throws Throwable
8:  {
9:    JAXBUtility.flushXmlToStream(
10:     "book.sample.dao.xml.po",
11:     order, new FileOutputStream(
12:         "PurchaseOrderGeneratedCementJ.xml"));
13:  }
14:}
```

Source: /src/book/sample/dao/xml/SampleJAXB.java

JAXB requires an XML schema as input to the code generator. If you haven't developed schemas for XML documents, you can use one of a number of tools available that derive schemas from XML documents. These will relieve you of the job of coding the schemas manually. I use a commercial tool called xmlspy (http://www.xmlspy.com/) to generate schemas from XML documents. You can find several other commercial and open source alternatives by doing a simple Web search.

You can run the JAXB code generator from a command prompt or an ANT script. Because the documentation contains several good examples of invoking the JAXB code generator from an ANT script, I won't duplicate that effort here. To invoke JAXB from a Windows command prompt, use the following:

```
java -jar %JAXB_HOME%/lib/jaxb-xjc.jar -d <output dir> -p <package name>
<schema>
```

For more information, refer to the JAXB documentation.

JAXB Usage Guidelines

The advantages of using JAXB are many. It drastically reduces the amount of custom application code needed for XML processing. JAXB-binded classes can easily be regenerated, so keeping up with XML document format changes is relatively easy. And because it's a native part of the JDK, you can be assured that JAXB will be around awhile.

JAXB is not appropriate for situations where you want to process selected tags of XML documents and ignore the rest. It doesn't provide any searching capabilities (as X/Path does). And it is not appropriate if you merely want to reformat an XML document into some other content type (such as HTML); I would use X/Path, XSL, and XSLT for these purposes.

Although most developers commonly use the marshaling and unmarshaling API portion of JAXB, I consider the API awkward. The designers of the API could have (and should have) written these actions as one-line calls to make them easier to use. I recommend using a shortcut, such as the one provided with CementJ, until less verbose ways to marshal and unmarshal become part of the JAXB API.

Never directly modify code generated by JAXB. If you do, those changes will be lost when you regenerate the binded classes for an XML document format change. Further, you would need to write test classes for these custom changes where no tests were necessary before.

If an XML schema is shared across applications, you might consider centralizing JAXB-generated classes. Most applications using an XML document don't require custom generation options and can use a centrally generated class set.

Generate JAXB into separate packages—don't commingle. If you commingle generated code with custom code, developers might have difficulty knowing what source is modifiable and what they shouldn't touch. In short, it's easier to enforce the "no modify" guideline previously described if you create separate packages.

Don't use generated JAXB classes as value objects in applications. In many applications, JAXB-generated interfaces resemble the value objects, implying that the JAXB-generated interfaces should replace the value objects.

However, exposing JAXB classes to the rest of your application makes your application vulnerable to changes in XML document format. Further, value objects often need capabilities that don't exist in the generated JAXB interfaces (e.g., `Serializable`, `Comparable`, overriding `equals()` and `hashcode()`, etc.). And the pain of using JAXB-generated classes in your application as value objects will persist long after you've forgotten the initial development time savings.

Using XSLT Within Java

XSLT and X/Path are often used in combination to reformat XML documents into a readable format, such as HTML or text. In essence, these technologies are used mostly to provide "reporting" capabilities for XML documents. Additionally, XSLT and X/Path can transform an XML document into any text format (e.g., another XML document or source code). This section assumes that you have a basic knowledge of XML, X/Path, and XSLT and focuses on how you can use them within the Java language. If you need a better understanding of these technologies, consult the tutorials at Zvon (http://www.zvon.org/).

XSLT is the mechanism by which XML data is transformed into HTML or text format. The details about this transformation are stored in an XSL style sheet. Listing 11.8 is a sample style sheet for the `<purchase-order>` example we've been using. This style sheet produces an HTML page listing all customer orders and ordered items.

Listing 11.8: Sample XSL Style Sheet

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="customer-order">
  <p>
    Order Nbr: <xsl:value-of select="@order-id" />
    Date Created: <xsl:value-of select="@date-created" />
    Date Shipped: <xsl:value-of select="@date-shipped" />
  </p>
  <xsl:for-each select = "order-line">
    <li>
      Product: <xsl:value-of select="@product-id" />
      Quantity: <xsl:value-of select="@order-quantity" />
      Price: <xsl:value-of select="@order-price" />
    </li>
  </xsl:for-each>
```

```
</xsl:template>
</xsl:stylesheet>
```

Source: /xml/PurchaseOrder.xsl

In a layered architecture, you would want to perform this transformation in a DAO. The HTML output from this transformation would be returned to the presentation tier for display to a user. Listing 11.9 is an example of how you could do this within Java.

Listing 11.9: Initiating an XSL Transformation

```
 1:import javax.xml.transform.Transformer;
 2:import javax.xml.transform.TransformerException;
 3:import javax.xml.transform.TransformerFactory;
 4:import javax.xml.transform.stream.StreamResult;
 5:import javax.xml.transform.stream.StreamSource;
 6:
 7:public class SampleXSL
 8:{
 9:  public String runSimpleTransformation()
10:     throws TransformerConfigurationException,
11:        TransformerException
12:  {
13:    ByteArrayOutputStream output =
14:        new ByteArrayOutputStream (200000);
15:    TransformerFactory tFactory =
16:        TransformerFactory.newInstance();
17:    Transformer transformer = tFactory.newTransformer
18:        (new StreamSource("PurchaseOrder.xsl"));
19:    transformer.transform(
20:        new StreamSource("PurchaseOrder.xml"),
21:               new StreamResult(output) );
22:    return output.toString();
23:  }
24:}
```

Source: /src/book/sample/dao/xml/SampleXSL.java

XSLT Usage Guidelines

Don't format data as XML just to use XSLT reformatting capabilities. If the data to be reported isn't already in XML format, it usually isn't worth the trouble to put it into XML format solely to report from it via XSLT. A myriad of toolsets can generate reports directly from data in a relational database. Further, forming XML, parsing it, and transforming can be resource intensive.

Consider using the XSLTC compiler. If your style sheets are static (i.e., they aren't dynamically generated at runtime), use the XSLTC compiler (http://xml.apache.org/xalan-j/xsltc_usage.html). In most cases, this provides at least a 25 percent performance enhancement. As with JAXB-generated classes, put XSLTC-generated classes into a separate package structure and don't manually change them.

Internet Resources

Among the many Internet resources for XML, XSL, XSLT, X/Path, and other XML topics, the following are my favorites:

- ZVON.org (http://www.zvon.org/). The References and Tutorials sections of the site are great. The tutorials most relevant to this chapter are the X/Path Tutorial, the XSLT Tutorial, and the XML Schema Tutorial.
- W3Schools (http://www.w3schools.com/). Offers well-written and concise tutorials for XML and XSLT concepts as well as tutorials for other topics.
- Jeni's XSLT Pages (http://www.jenitennison.com/xslt/). Excellent, well-written tutorials that will get you up and running quickly.
- Jeni's Schema Pages (http://www.jenitennison.com/schema/).

Further Reading

Bradley, Neil. 2000. *The XSL Companion*. Reading, MA: Addison-Wesley.

12

Building Database Access Objects

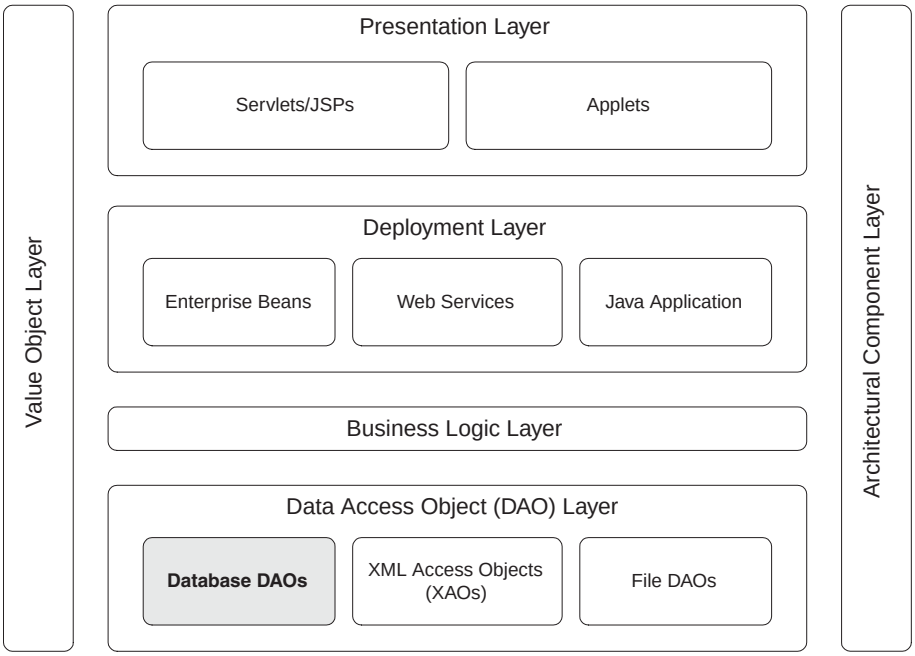
Data access objects read and write data in databases and convert that format to value objects usable by other layers in the application. For example, `PurchaseOrderDAO`, a DAO in a purchasing application, reads purchase order information from a database and converts it to value objects (e.g., `PurchaseOrderVO`) that the rest of the application can use. `PurchaseOrderDAO` also uses information in `PurchaseOrderVO` to update or insert data in the database. The following are some methods `PurchaseOrderDAO` might have:

```
public PurchaseOrderVO getPurchaseOrder(int orderNbr);
public void savePurchaseOrder(PurchaseOrderVO order);

public PurchaseOrderVO[] getCustomerPOs(String customerId);
public PurchaseOrderVO[] getUnshippedPOs();
public PurchaseOrderVO[] getBackOrderedPOs();
```

Figure 12.1 illustrates the role of DAOs in the software layer hierarchy.

All logic that interprets and processes that data is in the business logic layer, not in the data access object layer. The reason for segregating data access is to limit and consolidate your exposure to changes in the data source. For example, one of my clients migrated from using Sybase to Oracle. The migration was relatively easy in applications with segregated data access. Further, from a maintenance standpoint, it was easy to locate, modify, and

Figure 12.1: Using Data Access Objects Within a Layered Architecture



enhance the data access object layer to handle minor changes, such as column additions.

As discussed in chapter 5, most developers use native JDBC as a persistence mechanism. The current chapter provides some guidelines for using JDBC effectively and constructing JDBC data access objects that you can easily use in a layered architecture. For readers using entity beans or object-relational mapping tools, I illustrate how these technologies work in a layered architecture.

Data Access Object Coding Guidelines

The guidelines presented here apply only to custom-coded data access objects (entity bean users can skip to the next section). Listing 12.1 is an example of the guidelines in use.

Listing 12.1: Sample Data Access Object Code

```
 1:import book.sample.vo.PurchaseOrderVO;
 2:
 3:// some code omitted
 4:
 5:public class PurchaseOrderDAO
 6:{
 7:  private static final String SELECT_SQL =
 8:      "select CUSTOMER_ID, DATE_CREATED, DATE_SHIPPED " +
 9:      "   from    PURCHASE_ORDER " +
10:      "   where   ORDER_NBR = ?";
11:  public PurchaseOrderVO getPurchaseOrder(int orderNbr)
12:      throws SQLException
13:  {
14:    PurchaseOrderVO order = null;
15:
16:    PreparedStatement pStmt = null;
17:    ResultSet results = null;
18:
19:    try
20:    {
21:      OrderLineItemDAO lineDAO =
22:          new OrderLineItemDAO(this._dbConnection);
23:      pStmt = this._dbConnection.prepareStatement(
24:                  SELECT_SQL);
25:      pStmt.setInt(1, orderNbr);
26:
27:      results = pStmt.executeQuery();
28:      if (results.next())
29:      {
30:        order = new PurchaseOrderVO();
31:        order.setOrderNbr(orderNbr);
32:        order.setCustomerId(results.getString(
33:            "CUSTOMER_ID"));
34:        order.setOrderDate(results.getDate(
35:            "DATE_CREATED"));
36:        order.setShipDate(results.getDate(
37:            "DATE_SHIPPED"));
38:        order.setOrderedItems(
39:            lineDAO.getPOItems(orderNbr));
40:      }
41:      else
42:      {
43:    throw new DataNotFoundException
44:         ("Purchase order not found.  OrderNbr=" +
45:             orderNbr);
46:      }
```

```
47:     }
48:     finally
49:     {
50:         // CementJ alternative for close
51:           //-> DatabaseUtility.close(results, pStmt);
52:
53:           if (results != null)
54:           {
55:               try {results.close();}
56:               catch (SQLException s)
57:               {
58:                   // Log warning here.
59:               }
60:           }
61:         if (pStmt != null)
62:         {
63:             try {pStmt.close();}
64:             catch (SQLException s)
65:             {
66:                 // Log warning here.
67:             }
68:         }
69:     }
70:
71:     return order;
72:   }
73:}
```

Source: /src/book/sample/dao/db/PurchaseOrderDAO.java

Always pass the database connection from the business logic layer. How database connections are created is deployment specific. For example, you'll do a JNDI look-up for a database connection pool if the deployment is within a container, but you'll create one directly from the JDBC driver if the code is deployed as part of an application. DAO code should be application generic and usable in either place.

The code to make a database connection a settable property isn't complicated. The easiest way is to extend `DbDataAccessObject` (package `org.cementj.base`) from CementJ, which already contains connection-related code.

Database objects created within a method should be closed within the same method. Examples of this kind of object include `PreparedStatement`, `Statement`, `ResultSet`, `CallableStatement` objects. For many database platforms, it's necessary to close `ResultSet`, `PreparedStatement`, `CallableStatement`, and `Statement` objects. Some JDBC drivers close

these objects when the `Connection` is closed, and some don't. It's easier to spot bugs involving not closing an objects later if you adopt the convention of closing everything you create at this layer.

Because most JDBC objects throw a checked exception when they are closed, `close()` method calls must be enclosed in verbose try/catch logic. As you can see in listing 12.1, the finally block for the `SQLException` is very verbose. CementJ has a one-line convenience method to close JDBC objects. If you use the CementJ alternative, illustrated in line 51, you can omit lines 53 to 68.

CementJ logs errors on `close()` as a warning but does not produce an exception. If you want different behavior on an error from `close()`, you need to custom code that logic.

Do not close connection objects or issue commits, rollbacks, or savepoints within a DAO. Connection objects are created and closed in the business logic layer. Units of work, such as commits and rollbacks, are also handled in the business logic layer. Chapter 13 provides details on how to handle connection class creation, closing, and committing.

Using Entity Beans

For readers who use entity beans, this section illustrates how to incorporate them into a layered architecture. This is not intended as comprehensive entity bean coverage. For more in-depth information on using entity beans, see Alur et al. (2001).

Using the layered architecture as discussed throughout this book, entity bean usage occurs in data access objects and makes native JDBC code within data access objects unnecessary. In other words, each method in a DAO uses entity beans to retrieve and store data, not native JDBC. For example, consider a method on `PurchaseOrderDAO` called `getPOsForCustomer` that will look up all orders for a given customer. Listing 12.2 illustrates how the method uses an entity bean to retrieve information for the purchase orders and place them in `PurchaseOrderVO` objects so that they can be used throughout the application.

Listing 12.2: Using Entity Beans for Data Access Objects

```
1:  public PurchaseOrderVO[] getPOsForCustomer (
2:        String customerId)
3:  {
4:    PurchaseOrderVO[] po = null;
```

```
 5:    try
 6:    {
 7:      // Look up Entity Bean reference
 8:      Context ctx = new InitialContext();
 9:      PurchaseOrderHome poHome = (PurchaseOrderHome)
10:      PortableRemoteObject.narrow(
11:        ctx.lookup(
12:            "java:comp/env/BookSamples/PurchaseOrder"),
13:        PurchaseOrderHome.class);
14:      // -> End of Entity Bean Lookup
15:
16:      PurchaseOrder poEjb = null;
17:      Iterator poIt = null;
18:      Collection poC =
19:        poHome.findByCustomerId(customerId);
20:      if (poC.size() > 0)
21:      {
22:        po = new PurchaseOrderVO[poC.size()];
23:        int i = 0;
24:        poIt = poC.iterator();
25:        while (poIt.hasNext())
26:        {
27:            poEjb = (PurchaseOrder)poIt.next();
28:            po[i] = new PurchaseOrderVO();
29:            po[i].setCustomerId(poEjb.getCustomerId());
30:            po[i].setOrderNbr(poEjb.getOrderNbr());
31:            po[i].setOrderDate(poEjb.getDateCreated());
32:            po[i].setShipDate(poEjb.getDateShipped());
33:
34:            i++;
35:        }
36:      }
37:
38:    }
39:    catch (Throwable t)
40:    {
41:        throw new SampleException
42:              ( "Error searching PO. cust=" +
43:                customerId, t);
44:    }
45:
46:    return po;
47:  }
```

Source: /src/book/sample/dao/db/PurchaseOrderDAO.java

Note that this example uses container-managed persistence and local interfaces from the 2.0 EJB specification.

The example in listing 12.2 illustrates that in a layered architecture, DAOs

can have the same role within the application no matter what technology the objects use to interact with a database. Data access objects can be converted from using native JDBC to using entity beans (and vice versa) without affecting the rest of the application.

As a performance enhancement, you may want to consider placing the context and JNDI look-up (lines 7 through 14 of listing 12.2) in the constructor so that you don't repeat these operations for each call. I don't make this a general recommendation because it can cause availability issues in clustered environments.

A Hibernate Example

Object-relational toolsets appear to be gaining popularity. To illustrate how you can incorporate an object-relational toolset into a layered architecture, this chapter focuses on the Hibernate toolset, which is hosted at http://www.hibernate.org/.

Hibernate complements a layered architecture exceptionally well. The toolset requires mapping value objects to tables and columns in the database. With the mapping, Hibernate can read or write to the database using value objects directly. This greatly reduces the amount of Java code needed in DAOs.

Hibernate does have a configuration requirement. It's possible to specify this configuration in an XML file or by coding it in a central utility class. I elected to code Hibernate's configuration in a utility class called `HibernateEnvironment`, the source for which is shown in listing 12.3.

Listing 12.3: Sample Hibernate Configuration

```
 1:package book.sample.dao.db;
 2:
 3:import book.sample.vo.*;
 4:import book.sample.env.SampleException;
 5:
 6:import net.sf.hibernate.SessionFactory;
 7:import net.sf.hibernate.HibernateException;
 8:import net.sf.hibernate.cfg.Configuration;
 9:
10:import java.util.Properties;
11:
12:public class HibernateEnvironment
13:{
14:    private static SessionFactory _sessionFactory;
15:    static
```

```
16:     {
17:         Properties props = new Properties();
18:         props.put(    "hibernate.dialect",
19:             "net.sf.hibernate.dialect.Oracle9Dialect");
20:         props.put(
21:             "hibernate.cglib.use_reflection_optimizer",
22:             "true");
23:         props.put("hibernate.connection.driver_class",
24:             "oracle.jdbc.driver.OracleDriver");
25:         props.put("hibernate.connection.url",
26:             "jdbc:oracle:thin:@localhost:1521:ORA92");
27:         props.put(    "hibernate.connection.username",
28:                     "scott");
29:         props.put(    "hibernate.connection.password",
30:             "tiger");
31:         props.put(    "hibernate.connection.pool_size",
32:             "3");
33:         props.put(    "hibernate.statement_cache.size",
34:             "3");
35:
36:         Configuration cfg = new Configuration();
37:         try
38:         {
39:             cfg.addClass(OrderedItemVO.class);
40:             cfg.addClass(PurchaseOrderVO.class);
41:             cfg.setProperties(props);
42:
43:             _sessionFactory =
44:                 cfg.buildSessionFactory();
45:         }
46:         catch (HibernateException h)
47:         {
48:             throw new SampleException(
49:                 "Hibernate configuration error", h);
50:         }
51:     }
52:
53:     public static SessionFactory getSessionFactory()
54:     {
55:         return _sessionFactory;
56:     }
57:}
```

Source: /src/book/sample/dao/db/HibernateEnvironment.java

Configuration code such as that in listing 12.3 is only executed once when the application is started. It mainly consists of specifying the JDBC driver and connection information as well as registering each value object that can be persisted.

For each value object registered, a mapping must be created to tell Hibernate which fields in the class correspond to which columns in the database. Listing 12.4 is an example mapping for `PurchaseOrderVO`. Most of the content for this mapping can be easily generated by an open source Eclipse plug-in called Hibernator, which is available at http://sourceforge.net/projects/hibernator/.

Listing 12.4: Sample Hibernate Value Object Mapping

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class name="book.sample.vo.PurchaseOrderVO"
               table="purchase_order">
        <id name="orderNbr" column="ORDER_NBR"
               unsaved-value="0" >
          <generator class="hilo">
             <param name="table">purchase_order_nbr</param>
             <param name="column">next_value</param>
             <param name="max_lo">100</param>
          </generator>
        </id>
        <property name="customerId" column="CUSTOMER_ID" />
        <property name="orderDate" column="DATE_CREATED" />
        <property name="shipDate" column="DATE_SHIPPED" />
    </class>
</hibernate-mapping>
```

Source: /src/book/sample/vo/PurchaseOrderVO.hbm.xml

Hibernate has three classes to implement transaction management: `SessionFactory`, `Session`, and `Transaction`. `Session` manages JDBC connections for Hibernate, and as you might expect, `SessionFactory` is needed to establish a `Session`. `Session` objects produce `Transaction` objects that can be used for commits and rollbacks.

As when you use native JDBC, you handle all connection and transaction management logic in the business logic layer. Code illustrating how to use Hibernate `Session` and `Transaction` objects to establish a connection and commit a transaction is presented in listing 12.5.

Listing 12.5: Sample Hibernate Session and Transaction Management

```
1:import book.sample.dao.db.PurchaseOrderDAO;
2:import book.sample.dao.db.HibernateEnvironment;
3:
4:import net.sf.hibernate.Transaction;
5:import net.sf.hibernate.SessionFactory;
6:import net.sf.hibernate.Session;
7:
8:// Some code omitted
9:
10:        SessionFactory factory =
11:            HibernateEnvironment.getSessionFactory();
12:        Session session =
13:            factory.openSession();
14:        Transaction tx = session.beginTransaction();
15:
16:        PurchaseOrderDAO dao =
17:            new PurchaseOrderDAO(session);
18:        dao.savePurchaseOrder(order);
19:
20:        tx.commit();
```

Inside the DAO, the Hibernate session object is used to initiate selects, updates, and inserts. Listing 12.6 is an example of how to insert.

Listing 12.6: Sample Hibernate Insert

```
1:  public void savePurchaseOrder(PurchaseOrderVO order)
2:      throws SQLException
3:  {
4:    try
5:    {
6:        Integer generatedOrderNbr =
7:          (Integer) this._hibernateSession.save(order);
8:
9:        OrderedItemVO[] line = order.getOrderedItems();
10:       for (int i = 0 ; i < line.length; i++)
11:       {
12:           line[i].setOrderNbr(
13:               generatedOrderNbr.intValue());
14:           line[i].setLineNbr(i);
15:           this._hibernateSession.save(line[i]);
16:       }
17:   }
18:   catch (Throwable t)
19:   {
20:       throw new SampleException(
```

```
21:             "Error saving purchase order: " +
22:             order.describe(), t);
23:     }
24:   }
```

Source: /src/book/sample/dao/db/PurchaseOrderDAO.java)

JDBC Best Practices

Most J2EE applications manage their own persistence via JDBC, so a few tips and guidelines for JDBC usage are appropriate here. I assume that you already know the basics of JDBC programming. Readers wanting a good reference for JDBC basics as well as other programming topics should see Horstmann and Cornell (2001).

Use host variables in SQL statements instead of hard-coding literals in SQL strings. As a convenience, many developers embed literals in SQL statements instead. Listing 12.7 is an example of the bad practice of embedding literals. Notice that this example places a user ID directly in the SQL statement. Notice, too, that this example uses the + operator for string concatenation. Although using + is convenient, you can concatenate strings faster using `StringBuffers` and the `StringBuffer.append()` method.

Listing 12.7: Embedding Literals in SQL Statements (Bad Practice)

```
 1:    Statement stmt;
 2:    ResultSet rst;
 3:    Connection dbconnection;
 4:
 5:    // some code omitted
 6:
 7:    stmt = dbconnection.createStatement();
 8:    rst = stmt.executeQuery(
 9:        "select count(*) " +
10:            "from portfolio_info " +
11:            "where USER_ID = " +
12:            userID);
13:    if(rst.next())
14:    {
15:          count = rst.getInt(1);
16:    }
```

The problem with the code in listing 12.7 is that it circumvents database optimizations provided by Oracle, DB2/UDB, and many others. To get the benefit of database software optimizations, you need to use `PreparedStatement` objects instead of `Statement` objects for SQL that

will be executed multiple times. Further, you need to use host variables instead of literals for literals that will change between executions. With listing 12.7, the SQL statement for user ID 1 will be different than for user ID 2 (`"where USER_ID = 1"` is different from `"where USER_ID = 2"`). A better way to approach this SQL statement is presented in listing 12.8.

Listing 12.8: Using a Host Variable in a SQL Statement (Listing 12.7 Rewritten)

```
ResultSet rst;
PreparedStatement pstmt;
Connection dbconnection;
...
pstmt = dbconnection.prepareStatement ("select count(*) from
portfolio_info where USER_ID = ? ");
pstmt.setDouble(1,userID);
rst = pstmt.executeQuery();
if(rst.next())
{
 count = rst.getInt(1);
}
```

Notice that because listing 12.8 uses host variables instead of literals, the SQL statement is identical no matter what the qualifying user ID is. Further, a `PreparedStatement` is used instead of a `Statement`.

To better understand the database optimizations possible when using `PreparedStatement` objects, consider how Oracle processes SQL statements. When executing SQL statements, Oracle goes through the following steps:

1. Look up the statement in the shared pool to see if it has already been parsed or interpreted. If yes, go directly to step 4.
2. Parse (or interpret) the statement.
3. Figure out how to get the desired data and record the information in a portion of memory called the shared pool.
4. Get the data.

When Oracle looks up a SQL statement to see if it has already been executed (step 1), it attempts a character-by-character match of the SQL statement. If the program finds a match, it can use the parse information already in the shared pool and does not have to do steps 2 and 3 because it has done the work already. If you hard-code literals in SQL statements, the probability of finding a match is low (`"where USER_ID = 1"` is not the

same as `"where USER_ID = 2"`). This means that Oracle will have to reparse listing 12.7 for each portfolio selected. Had listing 12.7 used host variables and a `PreparedStatement`, the SQL statement (which would look something like `"where USER_ID =:1"` in the shared pool) would have been parsed once and only once.

DB2/UDB uses different terminology but a similar algorithm for dynamic SQL statements. Use of the `PreparedStatement` over the `Statement` is recommended for DB2/UDB as well.

Always close Statement, `PreparedStatement`, `CallableStatement` and `Connection` variables with a finally block. Many database platforms allocate resources to servicing these classes, and many continue to allocate those resources for a period if the objects aren't closed after use. Closing the variables improves time and resources spent on maintenance to keep errors from happening.

In the example shown in listing 12.9, a finally block closes `PreparedStatement`. The connection in the example method remains open because it is used elsewhere in the application. Also notice that the call to `DatabaseUtility` from CementJ closes the `PreparedStatement`. Using a utility to do the close makes it a one-liner. To execute `close()` on `ResultSet` and `PreparedStatement` directly, you need to use a try/catch to handle the `SQLException`.

Listing 12.9: Using a Finally Block to Close JDBC Resources

```
1:import org.cementj.util.DatabaseUtility;
2:
3://  Some code omitted..
4:private static final String GET_DISPLAY_SQL =
5:         "select t_ddlb_itm_dspl_val " +
6:         "from nasdb..nmr_parm_ddlb_val where " +
7:         "i_templ_parm = ? and c_ddlb_itm = ? ";
8:
9:public String getDisplayValue(int parmId,
10:    String parmValue)
11:  {
12:    String displayValue = null;
13:
14:    PreparedStatement pStmt = null;
15:    ResultSet results = null;
16:
17:    try
18:    {
19:      pStmt = _dbConnection.prepareStatement(
```

```
20:     GET_DISPLAY_SQL);
21:
22:       pStmt.setInt(1, parmId);
23:       pStmt.setString(2, parmValue.trim());
24:       results = pStmt.executeQuery();
25:
26:       if (results.next())
27:       {
28:         displayValue =
29:         results.getString("t_ddlb_itm_dspl_val");
30:         if (displayValue != null) displayValue =
31:              displayValue.trim();
32:       }
33:     }
34:     catch (Throwable t)
35:     {
36:       throw new MyApplicationRuntimeException(
37:             "Error selecting parm value::> parmId=" +
38:         parmId +
39:             "; parmValue=" + parmValue, t);
40:     }
41:     finally
42:     {
43:       DatabaseUtility.close(results, pStmt);
44:     }
45:
46:     return displayValue;
47:   }
```

Consolidate formation of SQL statement strings. As a former database administrator, I spend a substantial portion of my time reading the code others have written and suggesting ways to improve performance. As you might expect, I am particularly interested in the SQL statements. I find it especially hard to follow SQL statements constructed by string manipulation scattered over several methods. It greatly enhances readability if you consolidate the logic that forms the SQL statement.

Listing 12.10a is a good illustration of this point. Note that the string manipulation to form the SQL statement is located in one place. The SQL string is also defined statically to reduce the amount of string concatenation.

Listing 12.10a: Using a String Host Variable for a Date Field

```
Select sum(sale_price)
       From purchase_order
       Where to_char(sale_dt,'YYYY-MM-DD') >= ?
```

Limit use of column functions. Try to limit your use of column functions to the select lists of select statements. Moreover, use only aggregate functions (e.g., count, sum, and average) needed for select statements that use a "group by" clause. There are two reasons for this recommendation: performance and portability.

When you limit the use of a function to a select list (and keep it out of where clauses), you can use the function without blocking the use of an index. In the same way that the `to_char` function prohibited the database from using an index in listing 12.10a, column functions in where clauses likely will prohibit the database from using an index. This results in slower query performance. Rewriting the SQL statement as shown in Listing 12.10b allows most databases to use indices.

Listing 12.10b: Query with java.sql.Timestamp as a Host Variable

```
Select sum(sale_price)
       From purchase_order
       Where sale_dt >= ?
```

In addition, many of the operations for which developers use SQL column functions (data type conversion, value formatting, etc.) are faster in Java than if the database did them. I've had 5–20 percent performance improvement in many applications by avoiding some column functions and implementing the logic in Java instead. Another way to look at it is that you cannot tune column functions because you cannot control the source code. By implementing that logic in Java, you create code that you can tune if necessary.

Moreover, using non-ANSI standard column functions can also cause portability problems. Not all database vendors implement the same column functions. For instance, one of my favorite Oracle column functions, decode, which allows you to translate one set of values into another, is not implemented in many of the other major database platforms. In general, using column functions like decode has the potential to become a portability issue.

Always specify a column list with a select statement (avoid `Select *`). A common shortcut is to use the * in select statements to avoid having to type out a column list. Listing 12.11a illustrates this shortcut, and listing 12.11b illustrates the alternative of explicitly listing desired columns.

Listing 12.11a: Select Statement with *

```
Select * from customer
```

Listing 12.11b: Full Select Statement with Column List

```
Select last_nm, first_nm, address, city, state, customer_nbr from
customer
```

I recommend explicitly listing columns in select statements, as illustrated in listing 12.11b. The reason is that if someone reorders the columns in any of the tables in the select or adds new columns, the results obtained with the * shortcut will change, and you must modify the class. For example, suppose a database administrator changes the order of the columns, putting column `CUSTOMER_NBR` first (there are valid reasons why a database administrator would reorder columns) and adds a column called `COUNTRY`. The developer who used the * shortcut will have to change code: all the offset references used in processing the `ResultSet` will change. On the other hand, the developer who explicitly listed all the columns will be oblivious to the change because the code will still work.

Explicitly listing columns in a select statement is a best practice because it eliminates the need for maintenance in some cases.

Always specify a column list with an insert statement. Many developers use the shortcut of omitting the column list in an insert statement to avoid having to type out a column list. By default, the column order is the same as physically defined in the table. Listing 12.12a illustrates this shortcut, and listing 12.12b illustrates the alternative of explicitly listing desired columns.

Listing 12.12a: Insert Statement Without Column List

```
Insert into customer
       Values ('Ashmore','Derek','3023 N.Clark','Chicago',
              'IL', 555555)
```

Listing12.12b: Full Insert Statement with Column List

```
Insert into customer
       (last_nm, first_nm, address, city, state, customer_nbr)
       Values (?,?,?,?,?,?)
```

I recommend explicitly listing columns in insert statements, as illustrated in listing 12.12b, for the same reason you should explicitly list columns in

select statements. If someone reorders the columns in any of the tables in the insert or adds new columns, the insert could generate an exception, and you must modify the insert statement. For example, suppose that database administrator again changes the order of the columns, putting column `CUSTOMER_NBR` first, and adds a column called `COUNTRY`. The developer who used the shortcut will have to change code The developer who explicitly listed all columns will be oblivious to the change because the code will still work. In addition, note that listing 12.12b uses host variables, so the same `PreparedStatement` can be used for all inserts if there are multiple inserts.

As in select statements, explicitly listing columns in an insert statement is a best practice because it can eliminate the need for maintenance. Further, allowing reuse of the `PreparedStatement` improves performance, especially for inserts of large numbers of rows.

Code test cases for all DAO methods and put them in the test suite. You should be able to run a regression test for all objects in the data access layer at any time. This improves product quality by automating a reasonable test process.

Architect's Exercise: ProjectTrak

ProjectTrak uses exclusively custom-coded data access objects as described in this chapter and does not use entity beans. Many work-scheduling software packages are deployed as stand-alone applications on client workstations. Although these requirements are not documented in the use cases for ProjectTrak, we don't want the application design to make a stand-alone client deployment difficult later. Using entity beans requires the services of a J2EE container. All data access objects in ProjectTrak use CementJ `DbDataAccessObject` class and utilities.

The data model for ProjectTrak is given in figure 12.2. There are twelve entities. As no denormalization will be applied to this data model, all entities are implemented as tables. I always use the database to enforce referential integrity rules. The referential integrity rules highlight some types of errors at the data access level and force developers to fix them early. Data access is supported by the data access objects shown in figures 12.3a and 12.3b.

Figure 12.2: Data Model Diagram for ProjectTrak

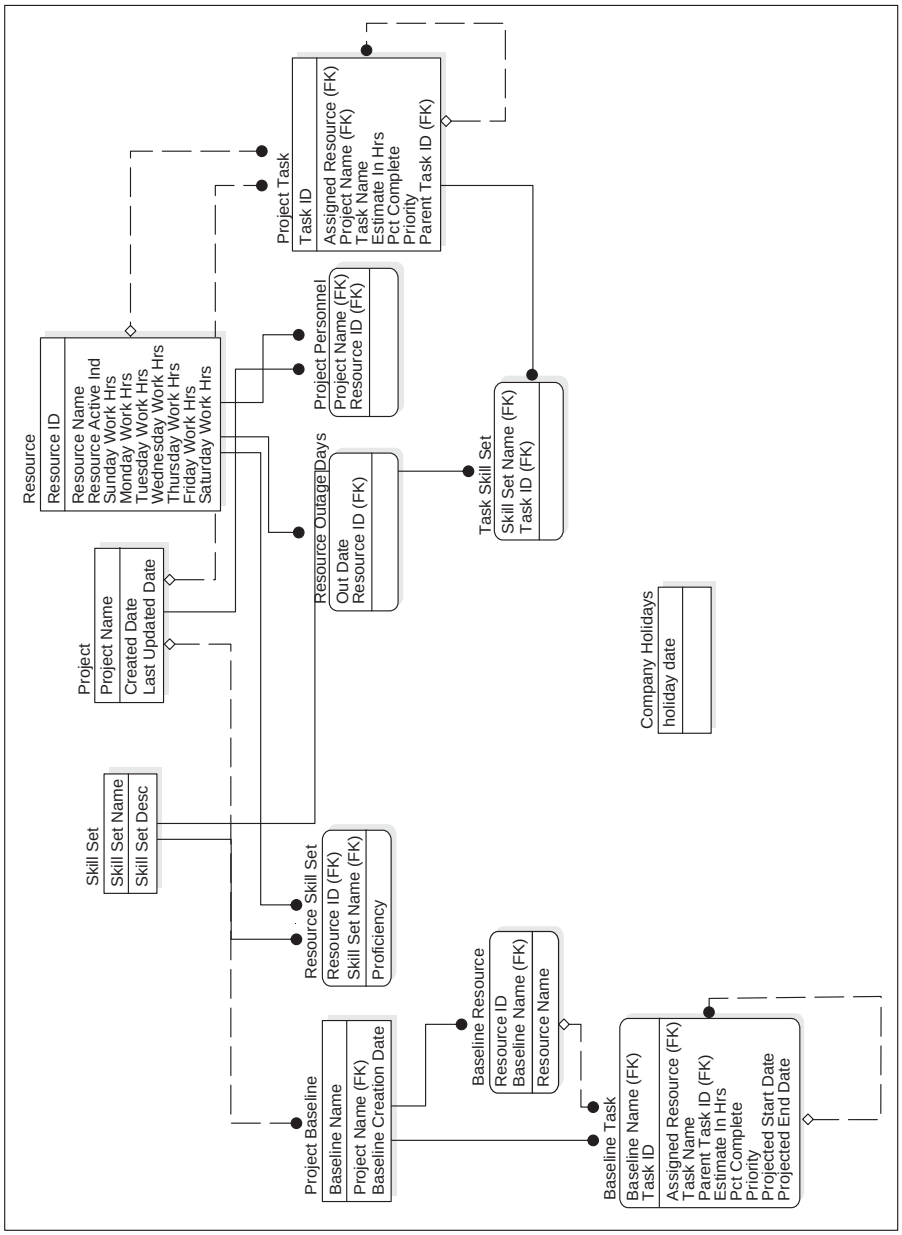


Figure 12.3a: Data Access Objects for ProjectTrak

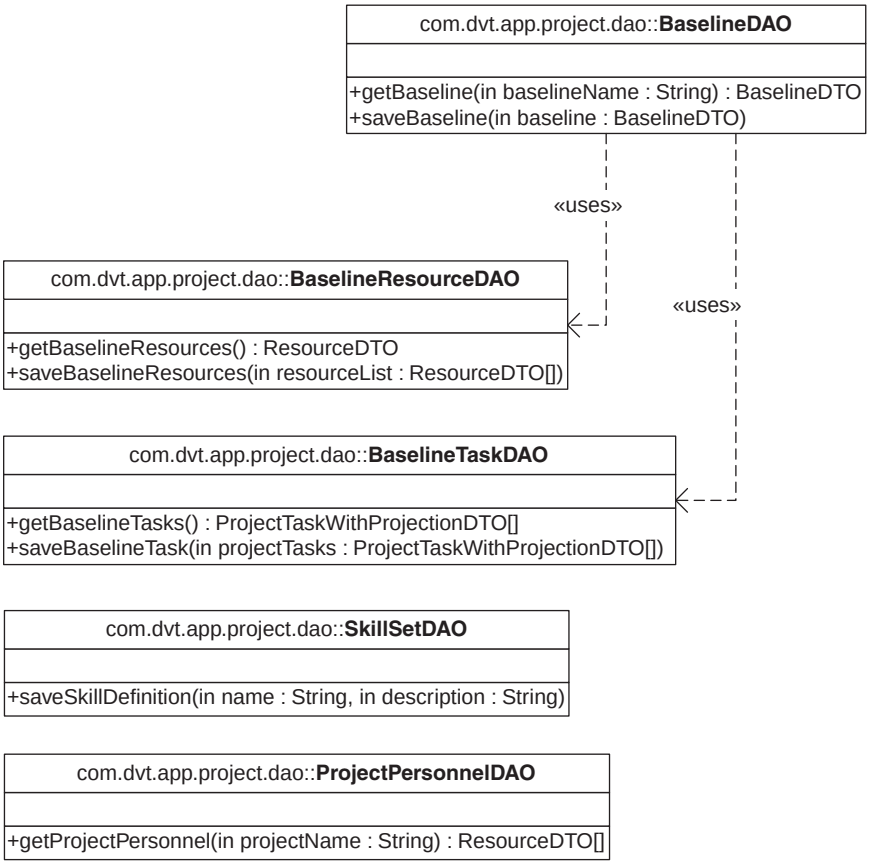
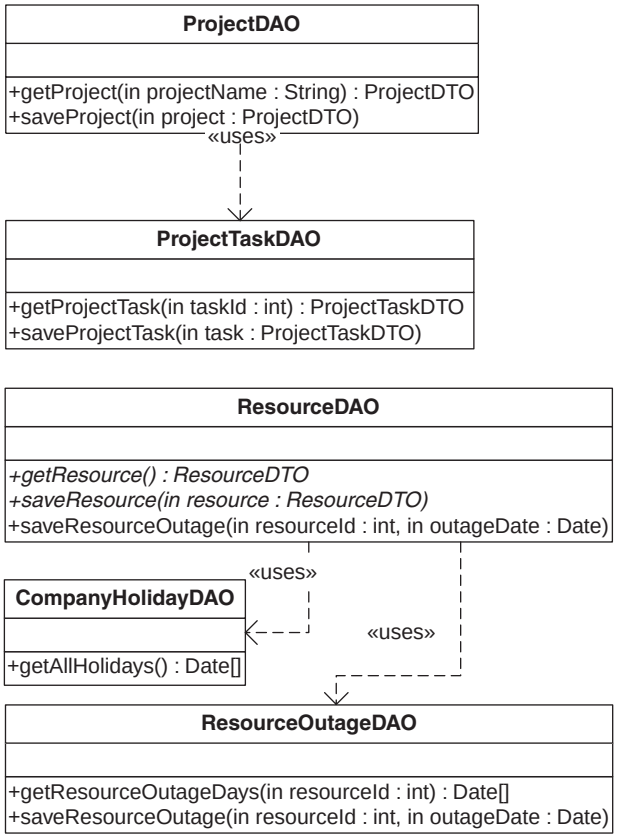


Figure 12.3b: Data Access Objects for ProjectTrak



Each entity is supported by a like-named data access object. For example, `ProjectTaskDAO` provides reading/writing capabilities for the `PROJECT_TASK` table. We might be tempted here to combine the reading/writing logic for multiple tables into a single data access object. Although this would reduce the number of data access objects, it doesn't really reduce the workload because the JDBC work is the most time consuming. Combining the data access objects just relocates where the work resides, it doesn't eliminate any work.

There is also value in consistency. Developers who weren't involved in the initial development will be able to find data access logic they need to work quickly and easily. If we combine access logic for multiple tables into a single data access object, it would take longer to find the logic we need to change.

Other Code Reduction Paradigms

JDBC coding is widely acknowledged as verbose. It isn't complicated code, but there tends to be a lot of it in most applications. Consequently, a number of products have been invented to reduce the amount of code (and the time spent on development and maintenance) for data access objects. These products have achieved mixed success in the marketplace. The bottom line is that they generally don't achieve enough code reduction to pay for their learning curve and performance overhead.

This section lists the most prevalent products and where to get more information about them. Although I don't use these products or recommend them, I think every technical architect should be aware of their existence.

Java Data Objects (JDO)

The JDO specification is now a formal part of the JDK (JSR-12). You can download the reference implementation and get more information at the JDO home page, http://java.sun.com/products/jdo/. Additional information about available JDO implementations can be found at http://www.jdocentral.com/.

CocoBase

CocoBase is a commercial object/relational mapping (ORM) product. You can download the software and obtain information on licensing from http://www.thoughtinc.com/.

TopLink

TopLink is a commercial ORM product that was acquired by Oracle Corporation. You can download the software as well as obtain the documentation from http://otn.oracle.com/products/ias/toplink/content.html.

OJB

The ObjectRelationalBridge (OJB) is an open source object/relational mapping tool from Apache. You can download the software and documentation from http://db.apache.org/ojb/.

Further Reading

Alur, Deepak, John Crupi, and Dan Malks. 2001. *Core J2EE Patterns: Best Practices and Design Strategies*. New York: Prentice Hall.

Horstmann, Cay S., and Gary Cornell. 2001. *Core Java 2, Volume II: Advanced Features*, 5th ed. Essex, UK: Pearson Higher Education.

Johnson, Rod. 2002. *Expert One-on-One: J2EE Design and Development*. Indianapolis, IN: Wrox Press.

13

Building Business Objects

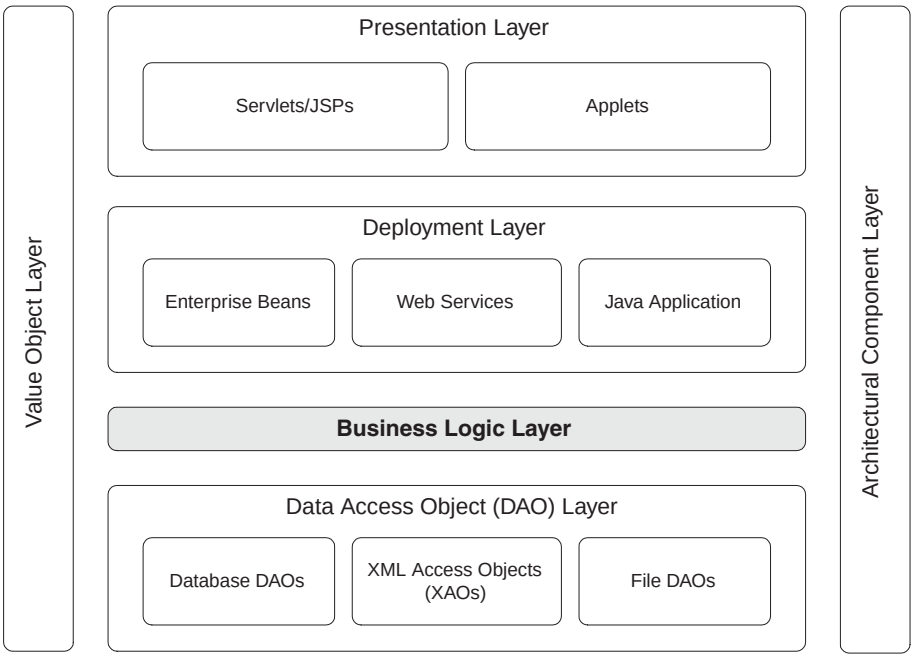
The business logic layer for J2EE applications combines data with business rules, constraints, and activities. I usually separate them from DAOs, VOs, and deployment wrappers, such as enterprise beans or Web services, to maximize the possibility of reuse. It's common for business objects (BOs) to use and coordinate the activities of multiple data access objects. Figure 13.1 illustrates how business objects function in a layered architecture.

Business objects are instantiated and invoked by other business objects or classes in the deployment layer, such as enterprise beans or Web services. They often instantiate and use classes in the data access layer, such as those discussed in the previous two chapters. This chapter provides coding guidelines used for business objects and presents examples taken from ProjectTrak.

As an illustration, consider a business object in a purchasing application. Class `PurchaseOrder` is in the business logic layer and is used by the deployment layer to provide the ability to view and submit purchase orders. `PurchaseOrder` has the following methods:

```
public PurchaseOrderVO getPurchaseOrderVO ();
public void setPurchaseOrderVO(PurchaseOrderVO po);
public void setPurchaseOrder (int ordered);

public void record ()
        throws InsufficientCreditException;
```

Figure 13.1: Using Business Objects Within a Layered Architecture




```
public void cancel ();

public PurchaseOrderVO[] getCustomerPurchaseOrders(
String custId );
```

Business objects are responsible for transaction management. Because business objects are the only classes that understand context, they should determine where transactions begin and when they are committed or rolled back. Business objects understand, for instance, which database inserts, updates, and deletes are necessary to perform a business function, such as adding a customer, defining a purchase order, or deactivating a retail product. These business functions can be saved together to form a composite transaction or can be issued individually.

Transaction Management

For J2EE applications, you can manage transactions differently than you do for other types of Java applications. Some texts refer to transaction management as transaction demarcation. The J2EE specification defines the Java

Transaction API (JTA), which provides a standard interface for transaction management. This standard interface allows container vendors to provide features such as two-phase commit capability.

J2EE developers typically have the choice of managing transactions programmatically via JTA or instructing the container (via enterprise bean deployment descriptors) to perform transaction management automatically. Many of my clients manage transactions programmatically, opting not to use JTA features and locally managing their transactions using JDBC. Programmed transaction management enables the business logic layer to function outside a J2EE environment.

Because of the variety of methods for managing transactions in J2EE applications, most texts advocate putting business logic in enterprise beans. They recommend this so you can take advantage of container transaction management features. Although I agree with the desire to offer advanced features, I respectfully disagree with the practice of making the business logic layer deployment specific.

Decouple transaction management tasks from the business logic layer with an interface. By decoupling transaction management, you can keep the business logic layer locally debuggable but still make advanced transaction management features available for J2EE deployments. An example of this concept is interface `TransactionContext` (from package `org.cementj.base.trans`), defined within CementJ.

`TransactionContext` is a part of CementJ, not a native part of the JDK or J2EE specification. If you prefer not to use CementJ directly, you can take this discussion as a "concept" example and still apply the concepts to the business logic layer. You would, of course, have to invent your own version of `TransactionContext`.

`TransactionContext`, as implemented in CementJ, is a transaction manager that provides database connections, begins transactions, and manages commits and rollbacks. Because `TransactionContext` is an interface, the implementation can be J2EE specific (i.e., use JTA) for J2EE deployments but use native JDBC for other types of environments. Either way, the business logic is identical in all environments.

Listing 13.1 contains the definition for `TransactionContext`.

Listing 13.1: Defining TransactionContext

```
1:package org.cementj.base.trans;
2:
```

```
 3:import java.sql.Connection;
 4:
 5:public interface TransactionContext
 6:{
 7:
 8:  public Connection getConnection(String label)
 9:      throws TransactionException;
10:
11:  public void commitAll()  throws TransactionException;
12:
13:  public void rollbackAll()
14:      throws TransactionException;
15:
16:  public void     begin()  throws TransactionException;
17:
18:  public void     closeAll() throws TransactionException;
19:}
```

Let's look at each method in detail. `TransactionContext` provides all database connections needed by the data access layer in the method `getConnection()`. Many applications use multiple database connections to different sources. For this reason, a "label" to identify the connection type is required. In a J2EE context, this will typically be your database pool name.

`TransactionContext` provides the ability to demark transaction beginnings using the method `begin()`. In a J2EE application, JTA usually performs this task. `TransactionContext` also provides a way to commit or roll back transactions using `commitAll()` and `rollbackAll()`, respectively.

`TransactionContext` enables you to close all opened connections via `closeAll()`. You should close all opened connections to prevent connection leaks, which we will discuss later.

It is imperative that you don't issue commits, rollbacks, or closes on the connections obtained from `TransactionContext` directly. Doing so will make your transactions "local" and circumvent the use of container transaction management features. Listing 13.2 illustrates the use of `TransactionContext`.

Listing 13.2: Using TransactionContext to Decouple Transaction Management

```
1:package book.sample.bo;
2:
3:import book.sample.vo.PurchaseOrderVO;
4:import book.sample.dao.db.PurchaseOrderDAO;
```

```
 5:// some code omitted
 6:
 7:public class PurchaseOrder
 8:    extends BusinessLogicObject
 9:{
10:
11:  // some code omitted
12:
13:  public void record()
14:     throws     InsufficientCreditException,
15:                 InternalApplicationException
16:  {
17:    if (_purchaseOrderVO == null)
18:       throw new IllegalArgumentException(
19:            "Null orders not allowed.");
20:
21:    try
22:    {
23:      this._transactionContext.begin();
24:      Connection conn =
25:          this._transactionContext.getConnection(
26:            "MyDbPoolName");
27:      CreditValidationBO creditBO =
28:          new CreditValidationBO(
29:              this._transactionContext);
30:      double availableCredit =
31:         creditBO.getAvailableCredit(
32:        _purchaseOrderVO.getCustomerId());
33:
34:      if (     _purchaseOrderVO.getTotalOrderAmount() >
35:               availableCredit)
36:      {
37:    throw new InsufficientCreditException
38:      (     "Sorry — your order exceeds your available" +
39:        " credit of $" + availableCredit + "!");
40:      }
41:
42:      PurchaseOrderDAO orderDAO =
43:          new PurchaseOrderDAO(conn);
44:      orderDAO.savePurchaseOrder(_purchaseOrderVO);
45:      this._transactionContext.commitAll();
46:    }
47:    catch (InsufficientCreditException i)
48:    {
49:      this._transactionContext.rollbackAll();
50:      throw i;
51:    }
52:    catch (InternalApplicationException iae)
53:    {
```

```
54:       this._transactionContext.rollbackAll();
55:       throw iae;
56:     }
57:     catch (Throwable t)
58:     {
59:       this._transactionContext.rollbackAll();
60:       throw new InternalApplicationException
61:             ( "Error recording PO ==> " +
62:               _purchaseOrderVO.describe(), t);
63:     }
64:     finally {this._transactionContext.closeAll();}
65:   }
66:}
```

Source: /src/book/sample/bo/PurchaseOrder.java

Business Object Coding Guidelines

Never put anything deployment specific in an object in the business logic layer. Business objects should be reusable in any deployment without changes. This insulates business logic from changes and developments in the deployment layer, which changes more rapidly than anything else. In the few years that Java has existed, an increasing number of distributed applications have used enterprise beans rather than CORBA. Over the past couple of years, Web services and message-driven beans have replaced entity beans. If you think J2EE and enterprise beans are the last programming evolution, remember the lesson of the Year 2000 problem: your code will probably live longer than you expect.

Many J2EE books guide you toward incorporating business logic directly into enterprise beans. I don't subscribe to that view because of the rapid pace at which deployment methodologies have changed over the years. Entity beans, if you use them, will have to be an exception. Use of entity beans definitely limits how your application can be deployed.

Always accept environment resources as arguments on construction. This includes `TransactionContext` and any additional JNDI resources. This advice is guided by the principle of keeping business logic deployment generic. It also tends to reduce complexity in the business logic layer. To simplify passing in commonly used resources, I usually extend `BusinessLogicObject` from CementJ (package `org.cementj.base`) for business objects, as illustrated in listing 13.2.

All public methods should explicitly validate the arguments. If you don't, you run a significant risk of generating a derivative exception, such as a

`NullPointerException`, which can take longer to debug and fix. If you generate an `IllegalArgumentException` with a clear message, programming errors in the deployment layer or within business objects that call you will have a better chance at being caught in unit testing.

Record the argument values in any generated exception. This practice makes internal system errors easier to replicate, thus easier to debug and fix. This is especially true if you've kept the business logic layer deployment generic and locally debuggable. Because most arguments are value objects, you can use the `describe()` method (if the value objects implement `Describable`) to do this quite easily. Listing 13.2 illustrates this practice.

A possible concern with catching `Throwable` is losing information contained within the root exception. However, the original exception and stack trace can be retained if your application exceptions extend either `ApplicationException` or `ApplicationRuntimeException` from CementJ or use the chained exception feature found in version 1.4 and above of the JDK.

Another common concern centers on the assumption that application code could not possibly be equipped to handle some of the exceptions that could get caught (e.g., `OutOfMemmoryError` or `ClassDefNotFoundError`). I argue that recording the circumstances and context surrounding such an error is necessary if developers are going to fix the problem. Recording these types of errors should not be left to chance. Chapter 17 provides a more extensive discussion of exception-handling practices and recommendations.

Any method that creates a database connection should close it in a finally block. If it doesn't, it will create a common problem for J2EE environments called a connection leak. A **connection leak** is a database connection that has been created but will never be closed. Most J2EE containers manage database connections by using connection pools. Connection pools are often configured to have a maximum size. If an application has allocated its maximum size, other users will err out. Thus a connection leak could inadvertently cause an error for another user. Also, connection leaks unnecessarily take up resources in the database server, as illustrated in listing 13.2.

Code test cases for all public methods of business objects and put the code in the regression test suite. All business objects can and should be testable individually. I like to put them in a regression test suite so that they can be easily run before new releases. I use JUnit for testing (http://www.junit.org). It's open source and easy to use.

Avoid using patterns not documented in one of the reputable pattern texts. I normally try to encourage creativity. However, at this point, hundreds of patterns have been identified and documented. The odds that you will encounter a requirement in a business application that isn't satisfied or addressed by at least one of the documented patterns is remote. As a reminder, I've listed several commonly used patterns in chapter 5.

Architect's Exercise: ProjectTrak

Given the somewhat simple use cases for the first release of the product, we've identified four publishable business objects. These objects are listed in the model shown in figure 13.2.

Of the four objects, the `ProjectBO`, `ResourceBO`, and `SkillSetBO` objects are mostly transactional in nature. They are primarily concerned with maintaining the integrity of data recorded in a relational database. The `TaskSchedulerBO` is more complex. This class does the work of determining the work schedule given the constraints that users have input.

Figure 13.2: Business Logic Layer Object Model

`TaskSchedulerBO` manages the process that derives the start and end dates for each task in a way that enforces the project's dependencies. `TaskSchedulerBO` will probably need more design work than what's documented in the model presented in figure 13.2.

Rather than drag the whole team through determining the work-scheduling algorithm and what additional support classes might be needed, the architect should work with a smaller group of developers (maybe even just one senior developer) to figure out the algorithm and then report back to the whole team. The more complex the problem, the longer it will take a large group to solve.

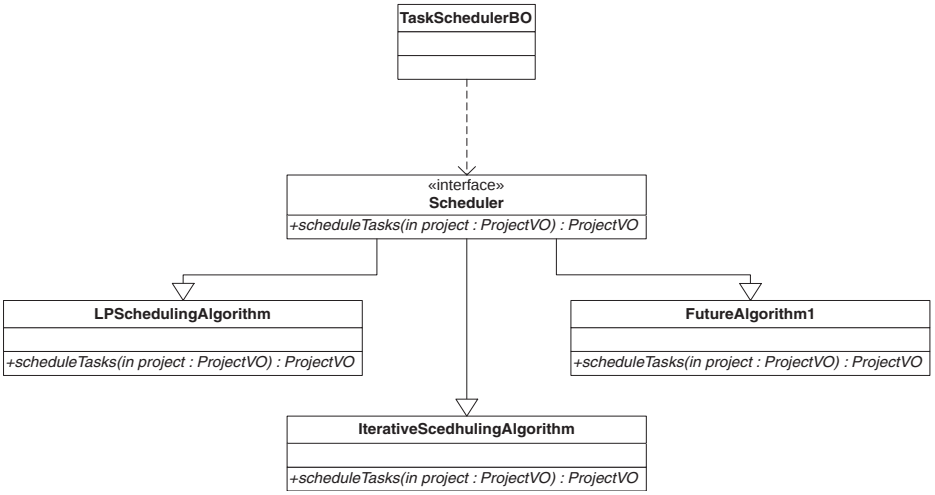
The use-case text that is causing us to write a task scheduler is the following:

- The system will compute work schedule information about project tasks. A projected start date and end date will be computed for each task. This date range will be consistent with the number of working hours available for the assigned resource. This projected date range will not conflict with the range generated for other tasks assigned to the resource.
- The order that tasks are completed will be consistent with their priority assignment.
- The order that tasks are completed will be consistent with the dependent tasks.

The use case tells us what the scheduler needs to do, but it really doesn't tell us how. This is as it should be, but it does give us a practical problem. I can think of several algorithms we can use to determine the schedule, but they will likely all produce a different result. For example, we could use an iterative algorithm that starts with the current day and iterates through, looking for free resources to assume tasks. Or we could use an algorithm that expresses these constraints as a classic linear programming problem. Linear programming is a branch of operations research that uses matrix algebra to solve optimization problems such as scheduling.

It is also highly likely that users will discover additional constraints after testing our scheduler and watching it do something they didn't agree with. After we change a scheduler, it would be handy to review the effects of the change on several projects to see if we're making the scheduler better or worse. Testers and developers will need an easy way to compare work schedules determined by different schedulers.

Figure 13.3: Sample ProjectTrak Strategy Pattern



Assuming for a moment that we could get users to agree to the additional use case that would provide users the ability to choose among several schedulers, it would be a prime candidate for the strategy pattern described in chapter 5. The algorithm candidates we have are different, but they all have the same inputs and outputs. Figure 13.3 is an example of applying the strategy model to ProjectTrak. The strategy pattern works by having `TaskSchedulerBO` rely on an interface, not a concrete class. We can then make the choice of scheduler data driven (e.g., from a user's choice).

Further Reading

Alur, Deepak, John Crupi, and Dan Malks. 2001. *Core J2EE Patterns: Best Practices and Design Strategies*. New York: Prentice Hall.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns.* Reading, MA: Addison-Wesley.

14

Building Deployment Layer Objects

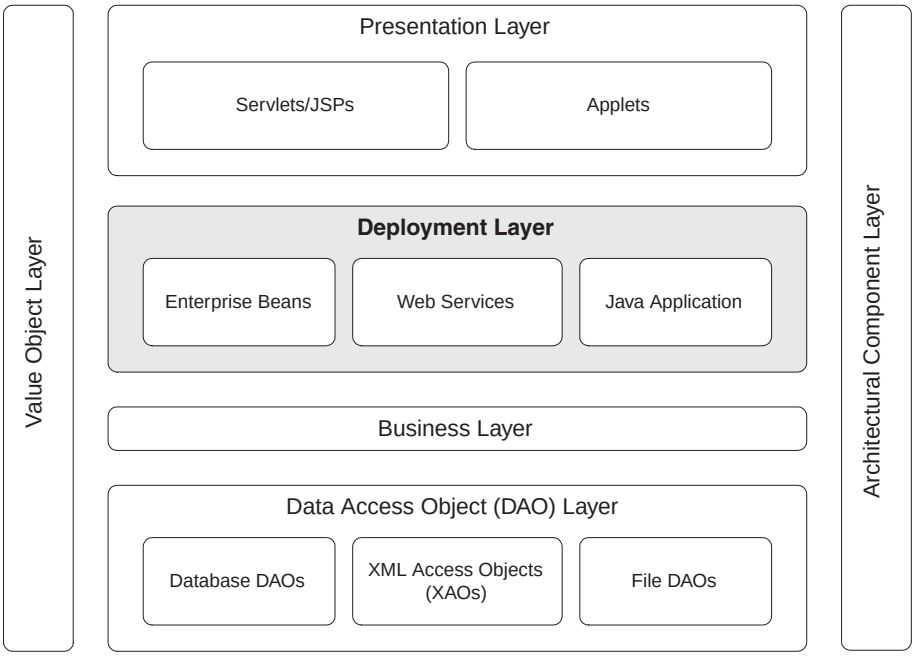
Objects in the deployment layer (which I call deployment wrappers) "publish" the content of the business logic layer to the presentation tier. In a J2EE world, the presentation tier can be on a separate server than the business logic layer. The deployment layer provides this remote calling capability. Deployment wrappers are purposely kept relatively simple and thin because they are more difficult and time consuming to develop and maintain than normal Java classes. Simple, thin deployment wrappers are less likely to have bugs.

Among the types of objects in the deployment layer are enterprise beans, Web services, and RMI services. The logic in the methods should contain the following activities:

- ▲ Instantiation/call to the underlying business object
- ▲ Initialization of the transaction context (defined in chapter 12)
- ▲ Log exceptions

Deployment wrappers, which are usually invoked from other applications or by elements of the presentation tier, should contain no business logic. Any business logic directly coded within a deployment wrapper would have to be replicated if you were to change your deployment strategy. The role of the deployment layer is depicted in figure 14.1.

Figure 14.1: Using Deployment Wrappers Within a Layered Architecture



Several deployment wrappers are available for J2EE applications. Session beans, message-driven beans, and Web services are among the most common. I focus on these three types of deployment wrappers in this chapter because the activities within them are strikingly similar. The chapter provides coding guidelines for each type and describes how to publish business objects using each type. Along the way, I'll point out additional resources in case you want more depth.

I've seen applications that call business objects directly from servlets, effectively making it a deployment wrapper. By so doing, they are effectively using servlets as deployment wrappers and eliminating the use of session beans or Web services. In other words, some applications run entirely within the servlet engine.

Session Beans

Session beans are a good choice for publishing functionality to multiple applications. In many environments, several applications run on separate

servers. By separating the application servers from where the servlet engine and the EJB container run, you can provide additional scalability and availability.

Session beans are also a good choice if you need tight coupling between two or more applications. Large organizations usually put applications in separate containers. If tight coupling between two applications exists (e.g., between an order entry and order fulfillment application), beans allow these applications to work in tandem yet be sized separately for the resources they need.

Because they are flexible and accessible by many different types of applications, session beans are also useful when the business object functionality is supporting multiple presentation tiers. I've worked on several applications in which the Web tier and administration tools were run as standalone applications. Enterprise beans allowed both applications to use the same services.

In a layered architecture where session beans publish but do not implement business logic, bean method signatures tend to be identical to method signatures in a business object. For example, consider listing 14.1, which is a session bean deployment for the `PurchaseOrder` class developed in chapter 12.

Listing 14.1: Sample Session Bean Deployment

```
 1:package book.sample.deploy.po;
 2:
 3:import book.sample.bo.PurchaseOrder;
 4:// some code omitted.
 5:
 6:public class PurchaseOrderBean
 7:    extends DefaultSessionBean
 8:{
 9:  public void recordPurchaseOrder(PurchaseOrderVO poVO)
10:     throws     InsufficientCreditException,
11:                 InternalApplicationException
12:  {
13:    try
14:    {
15:      J2EETransactionContext context =
16:         new J2EETransactionContext(
17:             this._sessionContext);
18:      PurchaseOrder po = new PurchaseOrder( context,
19:                                            poVO);
20:      po.record();
```

```
21:     }
22:     catch (InsufficientCreditException ice)
23:     {
24:         throw ice;
25:     }
26:     catch (InternalApplicationException iae)
27:     {
28:       LogManager.getLogger().logError(iae.getMessage(),
29:                                       iae);
30:       throw iae;
31:     }
32:     catch (Throwable t)
33:     {
34:       StringBuffer message = new StringBuffer();
35:       message.append("Error recording PO ==> ");
36:       if (poVO != null)
37:       {
38:           message.append(poVO.describe());
39:       }
40:       else message.append("null");
41:
42:       LogManager.getLogger().logError(
43:           message.toString(), t);
44:       throw new InternalApplicationException
45:             ( message.toString(), t);
46:     }
47:   }
48:}
```

Source: /src/book/sample/po/PurchaseOrderBean.java

The signature for an application method on the session bean corresponds to that of an object in the business logic layer. Remember that the bean isn't adding any business logic, it's just making a class in the busiess logic layer callable as an enterprise bean from remote machines.

Notice that listing 14.1 extends `DefaultSessionBean` instead of implementing the `SessionBean` interface directly. `DefaultSessionBean` is a convenience class in CemetJ that provides a default implementation for a session bean. I typically extend that to cut down the number of lines of code. If you do need to put logic in `ejbActivate()`, `ejbPassivate()`, or one of the other `SessionBean` methods, you can easily provide an override.

Deployment wrappers should perform error logging. The deployment wrappers are the last opportunity your application has to log before control is returned to a caller that might be on a remote machine. If your application will be the only code calling the session bean, it's possible to move logging

to the servlet or the Struts action class that calls the bean. Chapter 17 provides more detail on logging issues.

Whether it's better to use Log4J or the API logging provided in version 1.4 of the JDK is the topic of current debate. I typically decouple my loggers with interfaces. Similarly, CementJ implements a logging interface that can use either of these two logging packages and can easily be implemented for any other logging package you might want to use.

All exceptions thrown should implement `java.io.Serializable`. Otherwise, the client will likely receive a marshaling exception instead of the meaningful exception you tried to throw. In listing 14.1, both `InsufficientCreditException` and `InternalApplicationException` are serializable. Incidentally, if exceptions extend either `ApplicationException` or `ApplicationRuntimeException` from CementJ, they automatically implement `Serializable` as well as track the root exception with stack trace.

Using stateless session beans improve performance. A stateful session bean remembers information from a user's previous calls, whereas a stateless session bean does not. You should use stateful session beans if you're supporting multiple presentation tiers (e.g., HTML/JSPs, applets, and a heavy client-side application deployment). If your application requires you to maintain state, you won't want to replicate the state management logic in each presentation tier.

Stateless session beans should avoid instance-level variables. There's really no need for them in a stateless bean. A possible exception is storing the `SessionContext` provided by the container so that you have easy access to the `UserTransaction` for transaction management.

In an effort to be nice to my customers (callers), I provide a client implementation of the bean so they don't have to write what is in essence the same code. Listing 14.2 illustrates this concept.

Listing 14.2: Sample Session Bean Client

```
1:package book.sample.client;
2:
3:// some code omitted
4:
5:public class PurchaseOrderClient
6:{
7:  public PurchaseOrderClient() throws NamingException,
```

```
 8:                      CreateException,
 9:                      RemoteException
10:  {
11:    _controller = this.getController();
12:  }
13:
14:  public void recordPurchaseOrder(PurchaseOrderVO po)
15:     throws InsufficientCreditException,
16:        InternalApplicationException,
17:        RemoteException
18:  {
19:    _controller.recordPurchaseOrder(po);
20:  }
21:
22:  public PurchaseOrderVO[] getPOsForCustomer (
23:          String customerId)
24:    throws InternalApplicationException,
25:           RemoteException
26:  {
27:      return _controller.getPOsForCustomer(customerId);
28:  }
29:
30:  private PurchaseOrderController getController()
31:      throws     NamingException,
32:                  CreateException,
33:                  RemoteException
34:  {
35:    PurchaseOrderController controller = null;
36:    Context ctx = new InitialContext();
37:
38:    Object home =
39:        ctx.lookup("PurchaseOrderControllerHome");
40:    PurchaseOrderControllerHome controllerHome =
41:        (PurchaseOrderControllerHome)
42:              PortableRemoteObject.narrow(home,
43:                  PurchaseOrderControllerHome.class);
44:    controller = (PurchaseOrderController)
45:        PortableRemoteObject.narrow (
46:            controllerHome.create(),
47:            PurchaseOrderController.class);
48:
49:    return controller;
50:  }
51:
52:  private PurchaseOrderController _controller = null;
53:}
```

Source: /src/book/sample/client/PurchaseOrderClient.java

Message-Driven Beans

Message-driven beans (MDBs) allow the container to handle the threading associated with listening to a JMS queue. When MDBs are deployed, the container administrator provides information about what queue it listens for. When a message is received from a queue, the container allocates an MDB and calls the onMessage() method on it, providing the content of the message. Processing within the MDB is similar to a stateless session bean in many ways. MDBs have the same set of responsibilities that session beans do, with a few differences. For the purchase order example, consider the onMessage() method in listing 14.3.

Listing 14.3: Using an MDB to Process Purchase Orders

```
 1:package book.sample.deploy.poxml;
 2:
 3:import book.sample.bo.PurchaseOrder;
 4:// some code omitted
 5:
 6:public class PurchaseOrderMessageDrivenBean
 7:    extends DefaultMessageDrivenBean
 8:{
 9:
10:  public PurchaseOrderMessageDrivenBean() {}
11:
12:  public void onMessage(Message message)
13:  {
14:    String xmlText = null;
15:    PurchaseOrderVO order = null;
16:
17:    try
18:    {
19:      J2EETransactionContext context =
20:         new J2EETransactionContext(
21:             this._messageDrivenContext);
22:      if (    message != null &&
23:              message instanceof TextMessage)
24:      {
25:        TextMessage tm = (TextMessage) message;
26:        xmlText = tm.getText();
27:      }
28:      else
29:      {
30:        LogManager.getLogger().logError
31:              ("Null or invalid message received by " +
32:             "PurchaseOrderMessageDrivenBean");
33:      }
```

```
34:
35:       order = this.customerOrderListXlator(xmlText);
36:
37:       PurchaseOrder po = new PurchaseOrder(    context,
38:                                                order);
39:       po.record();
40:     }
41:     catch (InsufficientCreditException ice)
42:     {
43:       LogManager.getLogger().logInfo(ice.getMessage(),
44:                                      ice);
45:     }
46:     catch (InternalApplicationException iae)
47:     {
48:       LogManager.getLogger().logError(iae.getMessage(),
49:                                       iae);
50:     }
51:     catch (Throwable t)
52:     {
53:       StringBuffer errMessage = new StringBuffer();
54:       errMessage.append("Error recording PO ==> ");
55:       if (order != null)
56:       {
57:           errMessage.append(order.describe());
58:       }
59:       else errMessage.append("null");
60:
61:       LogManager.getLogger().logError(
62:           errMessage.toString(), t);
63:     }
64:     finally
65:     {
66:       JMSUtility.acknowledgeMessage(message);
67:     }
68:   }
69:}
```

Source: /src/book/sample/poxml/PurchaseOrderMessageDrivenBean.java

The MDB deployment for recording a purchase order has the additional burden of interpreting the message, but otherwise processing is very similar to a session bean. I typically use XML as a protocol for messages. However, this is a preference, not a technical requirement.

MDBs should not throw exceptions. There's no application "caller" to throw the exception to. The best you can do is log the error and possibly e-mail or page an application administrator. If you do throw an exception, it would

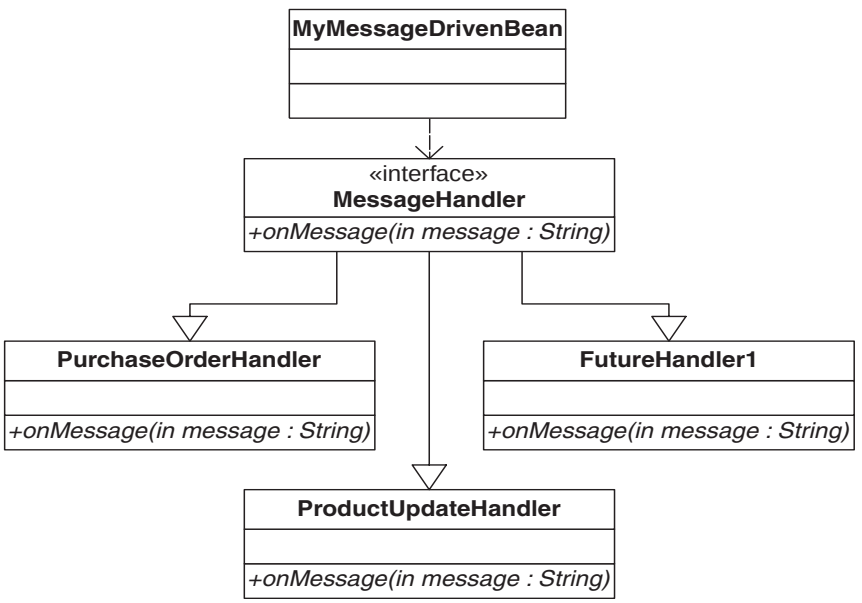
most likely be recorded in the containers' logs. My experience is that in most organizations, the probability that anyone will see it is extremely low.

Acknowledge message receipt in a finally block. If you receive a message but don't acknowledge it, it may be redelivered. This leads to a common messaging problem known as a "poison message." Essentially, if message processing continually errors out without acknowledgment, it creates an infinite loop when the message is redelivered. Generally, you should acknowledge the message even if processing the content of the message produced an error. Acknowledging the message means only that you successfully received it.

An annoying consequence of message acknowledgment is a checked exception. To avoid this, I use a convenience class from CementJ that reduces the acknowledgment to one line of code as illustrated on line 66 of listing 14.3.

You can send multiple types of messages over one queue. For example, you might send customer information updates, product updates, and purchase orders over the same queue. In this case, the MDB would not directly contain logic to process the message but would forward it on to a handler. Figure 14.2 is an object model illustrating this concept.

Figure 14.2: Sample MDB Receiving Multiple Message Types



Web Services

All the major container vendors are providing ways to configure stateless session beans so they can be called as a Web service using the SOAP protocol. This is the easiest and fastest way by far for you to get a Web service up and going quickly, because all it requires on the server side is a configuration change. The guidelines given for session beans earlier in the chapter apply to Web services directly.

Unfortunately, all the Web service client code that I've seen differs slightly for each SOAP service provider or vendor. Listing 14.4 is a sample using Apache, which appears to be popular, but you should consider this a "concept" example that you may not be able to take literally.

Listing 14.4: Sample Apache SOAP Web Service Client

```
 1:package book.sample.client.web;
 2:
 3:import book.sample.vo.PurchaseOrderVO;
 4:// some code omitted
 5:
 6:public class PoClient
 7:{
 8:    public PoClient() throws ServiceException
 9:    {
10:      _webService = new Service();
11:      _webServiceCall = _webService.createCall();
12:
13:      _webServiceCall.setTargetEndpointAddress(
14:          PO_SERVICE_URL);
15:    }
16:
17:    public void recordPurchaseOrder(
18:         PurchaseOrderVO order)
19:         throws     InsufficientCreditException,
20:                     InternalApplicationException,
21:                     RemoteException
22:    {
23:        QName recordPOQName =
24:            new QName("recordPurchaseOrder");
25:
26:        Object[] args = new Object[1];
27:        args[0] = order.describeAsXMLDocument();
28:        _webServiceCall.setOperationName(
29:            recordPOQName);
30:
31:        Object ret = null;
```

```
32:         ret = _webServiceCall.invoke(args);
33:     }
34:}
```

Source: /src/book/sample/client/web/PoClient.java

Notice that listing 14.4 passes XML text as an argument instead of as a value object directly. It certainly could have passed the value object directly. With Web services, passing complex data types is a little less straightforward than it should be, but it is possible. The primary reason to pass XML document text instead is performance.

Cohen (2003) has done some performance and scalability tests comparing various types of argument patterns for Web services. The practice of passing XML text as a string argument falls under the category of what he calls SOAP Remote Procedure Call Literal Encoding (SOAP RPC-literal). Compared with passing the value object directly, this type of code is easier to implement, is faster, and scales better.

Furthermore, passing XML text as a string argument is less sensitive to changes in SOAP vendors. Complex data types depend on serialization (and deserialization) techniques to encode content. Because vendors use different encoding techniques, the serializer on the client and server should be from the same vendor. Passing XML text (as string data) bypasses potential vendor-switching costs down the road.

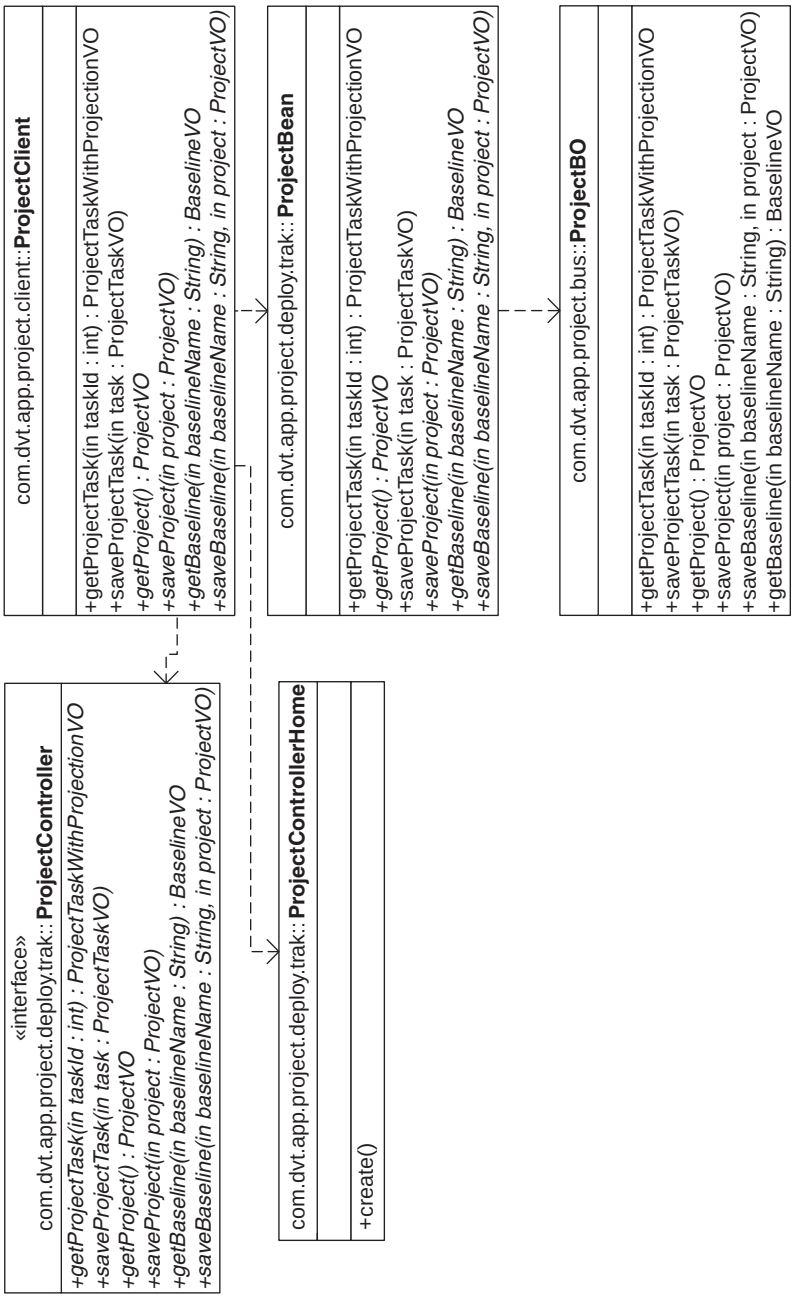
CementJ facilitates the practice of passing XML text in Web services. Value objects that extend `ValueObject` can easily be translated to XML text via the method `encodeAsXML()`. I would like to offer a `decodeFromXML()` in the future, but reconstituting a class from an XML document with enough reliability for production systems is a challenging task.

Architect's Exercise: ProjectTrak

In the previous chapter, we identified four business objects that provide functionality for a Web-compliant Java/J2EE presentation tier. At this point, the use cases don't require interfacing with additional clients or legacy systems. All processing identified is synchronous. Because this is intended as an enterprise-wide product, we would like to keep clustering technology capabilities for high availability.

Session beans are the J2EE deployment mechanism that meets these requirements. Neither Web services nor CORBA are required because we haven't identified a need to support non-J2EE platforms. Should this need come up in the future, we can add a deployment for it relatively easily. RMI services don't satisfy typical availability requirements for this kind of

Figure 14.3: Complete Object Model of the ProjectBean



application. And we don't require asynchronous processing or interfaces to legacy applications, so we don't need messaging technologies, such as JMS.

Given all of this, we'll deploy our business objects as session beans for the time being. With our layered architecture, we can add deployments or swap out our session beans for something else down the line if we need to.

I've adopted the shortcut discussed in chapter 5 of modeling the beans as one object. Each enterprise bean object in the model should be understood to contain a stateless session bean, a controller, a controller home, and a client stub. Figure 14.3 illustrates a more complete model for the `ProjectBean`, without that shortcut.

Further Reading

Cohen, Frank. 2003 (March). "Discover SOAP Encoding's Impact on Web Service Performance." *IBM DeveloperWorks*. Available online at http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/.

15

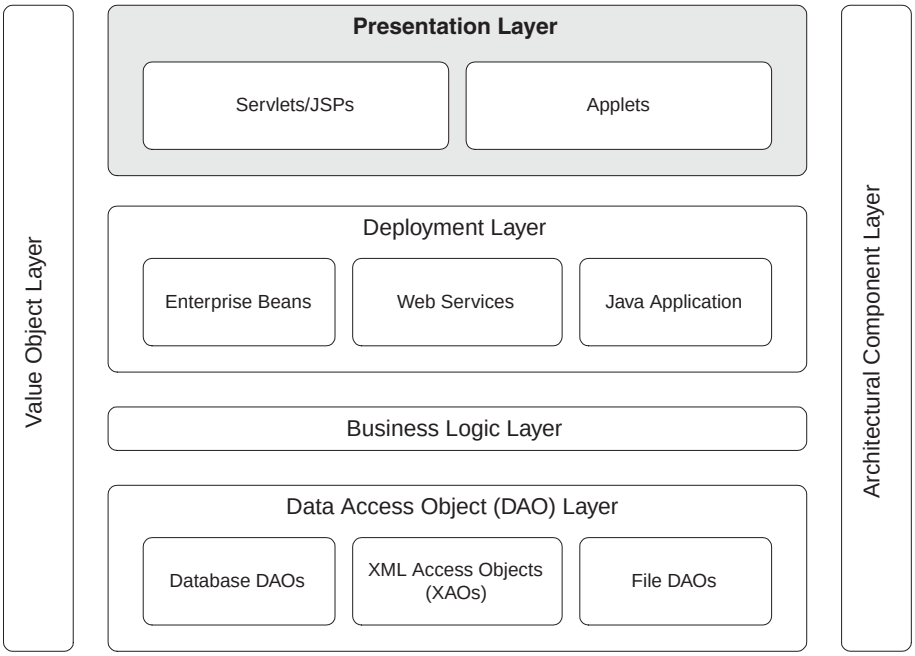
Building the Presentation Layer

Objects in the presentation layer produce the pages or screens that users physically see and use. There are several aspects to the presentation side: static content, dynamic content, and navigation. For J2EE applications, HTML pages typically provide static content, and a combination of JSPs and servlets usually provide dynamic content. For applications that require special effects (e.g., mouse-overs), Javascript can be used in combination with either static or dynamically produced HTML. Users navigate pages using controls (e.g., push buttons) on the page. Figure 15.1 depicts the role of the presentation layer.

Although it is technically possible to embed all presentation logic within JSPs, this type of architecture is difficult to maintain. A better method is to organize the presentation tier in a way that separates page display, user input validation, data processing, navigation, and security. This type of organization is often referred to as a model-view-controller (MVC) architecture. Although an MVC architecture results in more components to maintain, they are all relatively simple. For a more detailed discussion of the MVC pattern, see the Java Web site (http://java.sun.com/blueprints/patterns/MVC.html).

The most common framework used to organize the presentation layer (and support an MVC architecture) is Struts, the open source package from

Figure 15.1: Using Presentation Objects Within a Layered Architecture



Apache introduced in chapter 5 and available from http://jakarta.apache.org/struts/. As with many other frameworks, some of Struts' features are commonly used and others are rarely used in practice. I'll concentrate on the package's most commonly used features here to give you a basic understanding of how to use it to implement the presentation layer. Spielman (2003) is a good source for more in-depth information on Struts.

Some developers use XSL transformations to produce dynamic content, but the technique is not a formal part of the J2EE specification and produces mixed success. Although you might consider this option if your data exists in XML format, formatting data with XML merely to run transformations on it to produce HTML usually results in high development and maintenance costs.

Other types of presentation components are applets and Swing. These technologies are not J2EE specific and thus are out of the scope of this book. However, with a layered architecture, you certainly can deploy applets or Swing clients.

In a layered architecture, the presentation layer is not self-contained. Presentation objects rely on client stubs and the value objects used with those stubs in the deployment layer for dynamic content. You can change aspects of any other layer without having to change the presentation layer. Likewise, you can make changes in presentation without affecting the other software layers.

Presentation Components

Each component in the presentation layer has one of the following roles: page display, user input validation, data processing, navigation, or security. Guidelines for developing presentation layer components for each role are presented in the following sections.

Page Display

Most pages are either HTML for static content or JSPs for dynamic content. When using Struts, you should use JSPs only to create the page. JSPs are written with the assumption that the information they need to dynamically generate content has already been produced and associated with the `HttpSession`. With little in the way of conditional logic, JSP pages are relatively simple and easy to debug.

Consider an example from ProjectTrak. A JSP produces a page that allows users to view the information associated with project tasks (e.g., who's assigned to it, the percentage of the task completed, its name, etc.). Figure 15.2 illustrates what the page looks like.

Figure 15.2: JSP Page Output

ProjectTrak

Project Start Date: 07/01/03
Project Finish Date: 01/01/04

Assignments

Task ID	Task Name	Resource Name	Work	Start	Finish	% Work Complete
3	Screen Design and prototype	Derek Ashmore	16.0 hrs	07/28/03	07/30/03	0%
5	Base functionality	Derek Ashmore	16.0 hrs	06/26/03	06/27/03	0%
3	Screen Design and prototype	Derek Ashmore	16.0 hrs	07/28/03	07/30/03	0%
6	Skillset Tracking capability	Derek Ashmore	8.0 hrs	07/01/03	07/01/03	0%
7	Base functionality	Derek Ashmore	16.0 hrs	07/22/03	07/23/03	0%

The JSP assumes that information for a project task (the `ProjectTaskVO` object) has been retrieved and is already on the session. All the JSP has to do is obtain the value object from the session and populate the appropriate controls on the page. This is simple logic. Listing 15.1 has a code extract from the JSP.

Listing 15.1: JSP to Produce the Page in Figure 15.1

```
 1:<HTML>
 2:<!—
 3:  ProjectTrak Task Information Report
 4:
 5:    Author:    Derek C. Ashmore
 6:—>
 7:
 8:<%@ page import="com.dvt.app.project.dto.*,
 9:                 java.text.SimpleDateFormat;"
10:%>
11:
12:<jsp:useBean id="projectName" scope="session"
13:             class="java.lang.String" />
14:<jsp:useBean id="startDate" scope="session"
15:             class="java.lang.String" />
16:<jsp:useBean id="endDate"   scope="session"
17:             class="java.lang.String" />
18:<%
19:  ProjectTaskWithProjectionDTO[] task =
20:      (ProjectTaskWithProjectionDTO[])
21:          session.getAttribute("taskList");
22:%>
23:
24:<HEAD>
25:<TITLE><%= projectName %> Task Information</TITLE>
26:<META http-equiv="Content-Type"
27:      content="text/html; charset=windows-1252">
28:</HEAD>
29:
30:<BODY>
31:<H1>ProjectTrak</H1>
32:<P>
33:<BR>Project Start Date: <%= startDate %>
34:<BR>Project Finish Date: <%= endDate %>
35:<P>
36:<H2>Assignments</H2>
37:<TABLE BORDER>
38:  <TR BGCOLOR="#DFDFDF">
39:    <TH NOWRAP>Task ID</TH>
```

```
40:     <TH NOWRAP ALIGN=left>Task Name</TH>
41:     <TH NOWRAP ALIGN=left>Resource Name</TH>
42:     <TH NOWRAP>Work</TH>
43:     <TH NOWRAP>Start</TH>
44:     <TH NOWRAP>Finish</TH>
45:     <TH NOWRAP>% Work Complete</TH>
46:   </TR>
47:
48:   <%
49:       SimpleDateFormat format =
50:           new SimpleDateFormat("MM/dd/yy");
51:       for (int i = 0 ; i < task.length ; i++)
52:       {
53:   %>
54:
55:     <TR BGCOLOR="#FFFFFF" ALIGN=right>
56:       <TD ALIGN=center><%= task[i].getTaskId() %></TD>
57:       <TD ALIGN=left><%= task[i].getTaskName() %></TD>
58:       <TD ALIGN=left>
59:<%= task[i].getAssignedResource().getResourceName() %>
60:       </TD>
61:       <TD NOWRAP>
62:         <%= task[i].getEstimateInHours() %> hrs
63:       </TD>
64:       <TD NOWRAP>
65: <%= format.format(task[i].getProjectedStartDate()) %>
66:       </TD>
67:       <TD NOWRAP>
68: <%= format.format(task[i].getProjectedEndDate()) %>
69:       </TD>
70:       <TD>0%</TD>
71:     </TR>
72:
73:   <%
74:       }
75:   %>
76:
77:</TABLE>
78:</BODY>
79:</HTML>
```

Source: /jsp/TaskList.jsp

A key to successful JSP development is using them for content display only. JSPs with navigation and business logic embedded in them can be complex and difficult to debug. Debugging complicated JSPs is a time-intensive effort, and without the tools required for interactive debugging, the developer is reduced to primitive tracing.

User Input Validation

User input validation (e.g., ensuring the user entered all required fields) can be performed on the client with Javascript or on the server. In a Struts world, server-side user validation is delegated to `ActionForm` classes. Client-side input validation is often faster, especially if the user has a slow Internet connection. However, because support for Javascript support varies from browser to browser, Javascript can be a maintenance nightmare. These maintenance issues and the proliferation of high-speed Internet connections support validating user input on the server side.

`ActionForm` objects aren't required for Struts unless you're doing server-side input validation. If you've nothing to validate for a page, don't bother creating a form for that page.

Struts offers two ways to manage input validation. The first is to use the Struts Validator plug-in, which allows you to essentially program the validation rules in XML documents instead of Java. Although some developers see this as being easier, I don't. The set-up overhead and complexity make this an unattractive option in my view. For more detail on this point, see the Struts documentation.

Another alternative is to code the validation rules in Java within the form. Validation rules are classes that extend `org.apache.struts.action.ActionForm` and are coded in an override to method `validate()`. The `validate()` method returns object `ActionErrors`, which contains a description of all errors found. Struts manages navigating users to the pages they came from, and the JSP takes care of displaying validation errors, should they be present. Listing 15.2 illustrates the `validate()` override in a form. This option sounds straightforward enough, but there's more.

Listing 15.2: Sample Form Containing Validation Rules

```
 1:  public ActionErrors validate( ActionMapping mapping,
 2:                HttpServletRequest request)
 3:  {
 4:    ActionErrors errors = null;
 5:
 6:    if (_proj == null)
 7:    {
 8:      errors = new ActionErrors();
 9:      errors.add("proj",
10:          new ActionError("Null project not allowed."));
11:     }
12:    else if (_proj.equals(""))
```

```
13:     {
14:       errors = new ActionErrors();
15:       errors.add("proj",
16:           new ActionError("Blank project not allowed."));
17:     }
18:
19:     if (errors != null)
20:     {
21:         request.getSession().setAttribute("errors",
22:                                           errors);
23:     }
24:     return errors;
25:   }
```

It turns out that `ActionError` objects contain a key that is used to look up message text in a properties file, which is managed as a `ResourceBundle`. Using `ResourceBundle` objects, you can support user messages in multiple languages. This powerful feature is a necessary complexity for multinational applications. However, managing and coordinating these keys between your properties files and your code is annoying and painful, so if your business applications aren't written to support multiple languages, don't add the unnecessary complexity.

Struts provides a tag library to assist JSPs in displaying validation errors should they occur. These libraries can be used with either of the two validation methods previously described.

Notice that listing 15.2 doesn't define the `ActionError` messages with property keys but inserts the message text instead. It also stores the error messages on the session, which isn't typically required when overriding `validate()`. This is an inelegant shortcut I often use with Struts.

In JSPs that require input validation, you can insert a one-line include statement like the following:

```
<jsp:include page="/jsp/ShowErrors.jsp" flush="true"/>
```

With the JSP include statement, you can retrieve error messages from the session and format them for the user. Because it doesn't use the `ActionError` objects as they we're intended to be used, this shortcut is crude. However, it does have the effect of eliminating the overhead of managing any property files, thus saving you development time. Listing 15.3 illustrates `ShowErrors.jsp`. As applications vary widely in look and feel, you'll want to customize the error display to fit the look and feel of each application.

Listing 15.3: Using ShowErrors.jsp to Validate User Input

```
<%@ page import="org.apache.struts.action.ActionErrors,
                 org.apache.struts.action.ActionError"
%>

<jsp:useBean id="errors" scope="session"
    class="org.apache.struts.action.ActionErrors" />

<%
  if (errors != null)
  {
    java.util.Iterator errorIt = errors.get();
    ActionError error = null;
    while (errorIt.hasNext())
    {
        error = (ActionError) errorIt.next();
%>
      <li>
          <font color="red"><%= error.getKey() %>
          </font>
      </li>
<%
     errors.clear();
     session.removeAttribute("errors");
    }
  }
%>
```

Source: /jsp/ShowErrors.jsp

Data Processing

After validating user input, Struts delegates all processing to action classes that extend `org.apache.struts.action.Action`. It's common to override the method `execute()`, which usually does all the processing. Instead, an action class uses parameters on the request (or information already on the `HttpSession`) as arguments for a call to something in the deployment layer. The deployment layer component returns information that the action class puts on the session.

In ProjectTrak, one of the use cases requires a list of task assignments associated with a project. The action class that ProjectTrak uses to produce the task list is `ProduceTaskListAction`. It instantiates a `ProjectClient`, which was discussed in the last chapter, and invokes the `getProject()` method that retrieves a `ProjectVO` containing all the information the JSP needs for display. The logic within the action class is relatively simple. Little

conditional logic is required. Listing 15.4 is an extract of code from `ProduceTaskListAction`.

Listing 15.4: Using ProduceTaskListAction to Process Data

```
 1:package com.dvt.app.project.action;
 2:
 3:import com.dvt.app.project.client.ProjectClient;
 4:import com.dvt.app.project.vo.ProjectVO;
 5:
 6:import javax.servlet.http.HttpServletRequest;
 7:import javax.servlet.http.HttpServletResponse;
 8:import org.apache.struts.action.*;
 9:import java.text.SimpleDateFormat;
10:
11:public class ProduceTaskListAction extends Action
12:{
13:
14:  public ActionForward execute( ActionMapping mapping,
15:                ActionForm form,
16:                HttpServletRequest request,
17:                HttpServletResponse response)
18:         throws Exception
19:  {
20:    ActionForward forward = null;
21:    SimpleDateFormat format =
22:        new SimpleDateFormat("MM/dd/yy");
23:    ProjectClient projectClient = new ProjectClient();
24:    ProjectVO projectDTO =
25:        projectClient.getProject(
26:            request.getParameter("proj"));
27:
28:    request.getSession().setAttribute("projectName",
29:        projectDTO.getProjectName());
30:    request.getSession().setAttribute("startDate",
31:        format.format(projectDTO.getProjectStart()));
32:    request.getSession().setAttribute("endDate",
33:        format.format(projectDTO.getProjectEnd()));
34:    request.getSession().setAttribute("taskList",
35:        projectDTO.getProjectTasks());
36:
37:    forward = mapping.findForward("success");
38:
39:    return forward;
40:  }
41:}
```

Navigation

With Struts, navigation is configured in the struts-config.xml file. You designate a URL mask (e.g., /trak/TaskEdit*) in the file to uniquely identify all task edit requests. Struts provides a controller servlet that understands, via the struts-config.xml file, that any request with this URL requires executing an action class and forwarding the request to a display URL.

In addition to the URL, the struts-config.xml file lists the action class and the URL of the display JSP (or static HTML page) to use once the action class is successfully executed. For example, struts-config.xml would designate `TaskDisplayAction` to be executed for each task edit request. The file would also specify that the request be forwarded to the display JSP to produce the HTML that will be sent to the browser. Listing 15.5 is an extract from a struts-config.xml file.

Listing 15.5: Using struts-config.xml for Navigation

```
<struts-config>
  <form-beans>
    <form-bean name="projectForm"
type="com.dvt.app.project.form.ProjectListForm"
    />
  </form-beans>
  <action-mappings>
    <action    path="/tasklist"
      type="com.dvt.app.project.action.ProduceTaskListAction"
      name="projectForm"
      scope="request"
      validate="true"
      input="/jsp/Project.jsp">
      <forward name="success"
        path="/jsp/TaskList.jsp"/>
      <forward name="failure"
        path="/jsp/ServerErrors.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

The struts-config.xml can also specify an error page if the action class doesn't process successfully.

Security

For most applications, the first step in security is establishing whether or not a user is supposed to have access. The question is usually decided by the Web server before the application is invoked. For most J2EE applications, if

a user successfully enters a user ID and password, the Web server forwards the user's HTTP(S) request to the application.

In many cases, an application is written to assume that if it was invoked, the user is entitled to the content. For example, if you subscribe to the online version of *BusinessWeek*, once you supply your user ID and password, you're entitled to the content. The BusinessWeek application doesn't need to know your specific identity.

Some applications have more sophisticated requirements, altering content based on the identity of the user. An example of this is an online *Wall Street Journal* subscription. Based on who you are (and the preferences you establish), any news regarding a specific list of companies you specify appears as "Company News" content.

Other applications alter content depending on user-specific groups. With J2EE applications, groups are more often referred to as roles. An example of this type of data access appears on the open source software development Web site SourceForge.net. SourceForge designates users as belonging to "projects." Within each project, users can be either an "admin" or a "developer." The options a user sees on SourceForge pages differ depending on the user's role affiliation.

If your application alters its content based on a user's ID or role, the presentation layer can obtain this information from the `HttpServletRequest` (from `javax.servlet.http`). The action classes and JSPs have access to the request. Given a variable `request` that is of type `HttpServletRequest`, the following line of code can get a user ID:

```
String userId = request.getUserPrincipal().getName();
```

I'm not aware of a standard way for a J2EE application to get a list of roles a user has access to, but it can easily verify a user's membership in a specific role. The following code validates that a user is in the "admin" role:

```
if (request.isUserInRole("admin"))
{
  // your application code here
}
```

Presentation Layer Coding Guidelines

Keep every `Action`, `ActionForm`, and JSP thin. They are difficult to debug if your company didn't buy tools that allow servlet debugging. You want as little conditional logic as possible. A layered architecture leads you to put all the complexity in the layers that are locally debuggable, like the business

and data access layers. If you use Javascript code, keep it thin for the same reasons.

Action classes and JSPs should not use instance-level variables. Otherwise, you have a good chance of getting incorrect behavior from your application because every instantiation of an action or JSP is used to service multiple users. In contrast, an `ActionForm` can have instance-level variables.

Common Mistakes

Putting business logic in JSPs or servlets. Business logic tends to make JSPs and servlets more complex and difficult to maintain unless your organization buys development tools that allow interactive debugging. I've seen Web applications with all server-side code embedded in servlets.

Declaring instance-level variables on servlets. Don't consider servlets thread safe by default (even if you implement `SingleThreadModel`). I've even seen a development team "synchronize" every method in the servlet in an attempt to compensate for the problems caused by declaring instance-level variables.

Using no formal navigation control. Without a formal control structure, presentation layer code is often unstructured and unorganized. It's usually harder to maintain. Besides Struts, another good presentation layer control package is Maverick (http://mav.sourceforge.net/).

Further Reading

Spielman, Sue. 2003. *The Struts Framework: Practical Guide for Java Programmers*. Boston: Morgan Kaufmann.

16

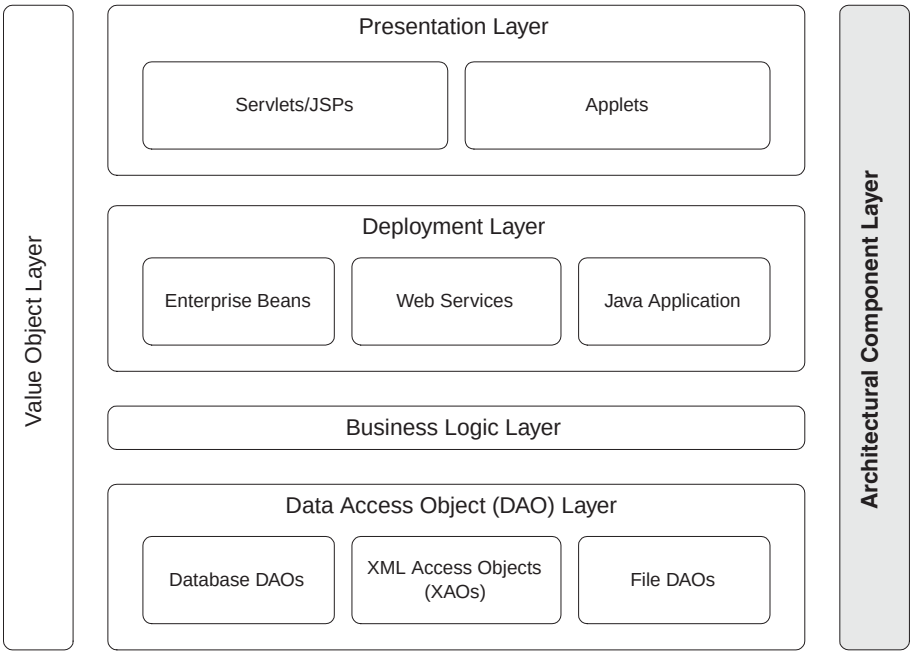
Building Architectural Components

Architectural components are classes and static utilities generic enough to be used for many applications. After you've identified a need for an architectural component, your first step should usually be looking in the marketplace for a component that meets that need. If you have the necessary budget, commercial alternatives are often better than building your own components. But I recommend starting your marketplace search by checking out open source alternatives, which can be just as good as commercial software and a lot easier on the budget. At the end of the chapter, I highlight some of my favorite open source Web sites.

One issue with component software, both open source and commercial, is the extreme variance in quality. Some components are easy to understand and use, others are inscrutable and impractical. In this chapter, I list the capabilities that developers generally associate with quality components. You can use this list to evaluate open source or commercial component software. I also present a series of tips and techniques for creating your own architectural components with those capabilities considered the marks of quality.

The role of architectural components is illustrated in figure 16.1.

Figure 16.1: Using Architectural Components Within a Layered Architecture



Component Quality

High-quality architectural components have the following capabilities:

- ▲ Shorten development time and effort
- ▲ Shorten expected maintenance time and effort
- ▲ Work in many applications because they are generic
- ▲ Work as advertised

These traits are desirable for any component software, whether it is commercial, open source, or authored by you.

A common guideline people use when judging PC software is: if they have to consult a manual to do basic tasks, they judge the software "too hard to use." You can use much the same standard when assessing component software. I consider component software difficult to use if:

- ▲ You have to read more than two pages of material to install and configure the component.

▲ It takes more than an hour from installing to using a component for a basic task.

Ease of use is the feature most commonly lacking in both commercial and open source component software.

A common mistake people use when designing architectural components is choosing a scope that is too large. A large scope usually results in a complex component that will take a long time to become available and have a long learning curve. I use the 80/20 rule when selecting features to implement in an architectural component. Component designers shouldn't get bogged down with features people will rarely use.

Making Components Easy to Use

If you're evaluating component software, ease of use should be one of the deciding criteria. If you're creating an architectural component, there are a number of things you can do (that many component developers don't do) to make your component easy to use.

Limit the instructions for basic tasks to a one-page cheat sheet. You can have a more detailed document for sophisticated, unusual tasks. The longer the material developers need to read to get going, the higher the percentage of developers who will give up in the process and look for something else.

Minimize the number of statements necessary to perform basic tasks. One of the chief benefits to using a component is eliminating code in the application. A one-line call is ideal because it eliminates typing and reduces the number of classes and methods you need to learn.

Consider the JAXB API as an example. A handful of lines must be issued to parse an XML document, as illustrated in listing 16.1.

Listing 16.1: Sample JAXB Parse

```
..... // some code omitted.
    InputStream xmlDocumentInputStream = new FileInputStream
                                        ( "PurchaseOrder.xml" );
    JAXBContext jc = JAXBContext.newInstance
                  ( "book.sample.dao.xml.po" );
    Unmarshaller u = jc.createUnmarshaller();
    CustomerOrderList order =
       (CustomerOrderList) u.unmarshal( xmlDocumentInputStream );
```

This series of calls could (and should) have been reduced to a one-line

call. Using CementJ, listing 16.2 performs the same parse with relatively simple code.

Listing 16.2: Sample JAXB Parse Using CementJ

```
..... // some code omitted.
CustomerOrderList order =
    (CustomerOrderList) JAXBUtility.getJaxbXmlObject
                        ("book.sample.dao.xml.po",
                         new File("PurchaseOrder.xml") );
```

The authors of the JAXP API could have provided a one-line call to transform an XML document. Instead, they force people to learn about `JAXBContext` and `Unmarshaller`, classes that contain features most developers rarely use.

One way to achieve a one-line call is to rely on a static method. This eliminates having to instantiate anything. The choice to make that call static is more than just a tactical choice. If you instantiated an object with the `getJaxbXmlObject()` method on it, you would not benefit from the fact that it was an object. For instance, you probably would not put the object in some type of collection or pass it between other objects as a method argument.

Minimize the number of arguments necessary to perform basic tasks. You can accomplish this by providing multiple overloads. Some of those overloads have small numbers of arguments with sensible defaults for the rest. For example, consider ThreadWorks, an API that makes multithreaded programming in Java easier and safer. Software and documentation for ThreadWorks are available at http://sourceforge.net/ projects/threadworks/.

The `TaskManager` from ThreadWorks hides the complexity of threading code, running tasks for you, on your behalf. A simple example is the following code, which asynchronously runs one or more tasks:

```
_taskManager.run(task);                 // Run One Task
_taskManager.run(taskCollection);       // Run several Tasks
_taskManager.run(taskArray);            // Run several Tasks
```

Optionally, you can run one or more tasks and have a `CompletionEventListener` execute when all are done, as follows:

```
_taskManager.run(task, completionEventListener);
_taskManager.run(taskCollection, completionEventListener);
_taskManager.run(taskArray, completionEventListener);
```

With `TaskManager`, it should be easy to perform a basic task, yet advanced capabilities can be made available.

Separate the classes meant for public consumption from those needed internally by the API. The more classes a component has, the longer it takes to find the class with the functionality you want. For example Struts' `org.apache.struts.action` package has three or four classes that are commonly used, and the rest are internal. Keeping all these classes together just adds to the time required to learn the API.

One way to solve this problem is to move classes not meant for public consumption to a separate package that's documented as "for internal use only." For example, ThreadWorks separates all internal classes into its `com.dvt.support` package. Users don't have to wade through low-level classes they don't need yet to find the functionality they want.

Provide samples that are easy to copy with an index. Make it easy to find a sample that is close to what the user wants. Most of us learn by example and don't type very quickly. Having something to copy from saves users time. A good place for short samples is within the JavaDoc.

Limit dependencies on other APIs. I once was forced to implement a poorly written scheduling component for a client (I wasn't given a choice). This component used two internal components that were hard to use and complex to configure. I've since learned how to avoid inflicting the same kind of pain on users of my open source components: use interfaces to decouple.

For example, CementJ depends on logging services in several places. Users wanting to try out the API should not have to configure logging services or use a specific logging package. With a logging interface that decouples, CementJ implements a console logger by default. Users can easily use Log4J or the logging package that comes with version 1.4 and above of the JDK. Alternatively, CementJ can be configured to use any logger.

Apache has a similar package for its open source Commons components. Called Logging, the package is a bit more complex and requires a bit more of a learning curve than CementJ. It can be downloaded from http://jakarta.apache.org/commons/logging.html.

Check all arguments on all methods meant for public consumption and produce clear error messages for invalid inputs. Rather than degenerating into derivative exceptions (e.g., null pointer exceptions), put information on how to correct problems in the exceptions. For example, "Invalid format

type argument" isn't as useful as "Invalid format type 'foo.' Valid types are the following constants on this class: PDF, HTML, XLS, and DOC." Simply displaying the erroneous value passed to a method might shorten the time it takes to debug and correct the issue.

Avoid throwing "checked" exceptions. Throwing "unchecked" exceptions, which extend `RuntimeException`, is preferred because it doesn't force the user into as much try/catch logic. Not needing as much code to use a component definitely makes it easier to use. For a more detailed discussion of this rather controversial concept, see chapter 17.

Making Components Easy to Configure and Control

Minimize the number of properties a user must configure. Some components use a properties file, which is a file of key and value pairs, to manage configuration settings. These files are often used in conjunction with the `java.util.Properties` object. Listing 16.3 is a short extract from a past release of the WebLogic™ application server software. It illustrates what a properties file looks like.

Listing 16.3: Sample Properties File

```
################################################
# Server configuration
################################################
# If the DTD location is changed from the default this property
# needs to changed to point to the new location.
commerce.xml.entity.basePath=H:\bea\weblogic700\portal/lib/dtd

################################################
# Logger class
################################################
commerce.log.class=com.beasys.commerce.util.NonCatalogLog
commerce.log.display.deprecated=true
commerce.log.display.debug=false
```

The more properties users need in order to choose values for running your component, the longer and harder your component is to configure. You can alleviate this problem by choosing sensible defaults for as many configuration properties as possible. In addition, clear error messages for incorrect configurations can make your components easy to use. The method BEA used for WebLogic™ was to write an installation program that configured required properties on install.

Minimize the required complexity of any needed XML documents. The more complex the document structure, the harder the component is to configure and control. Jakarta's Ant project is an excellent example of using XML files effectively. Ant is an XML scripting utility commonly used for application builds. Its scripting language is XML based, and its structure is simple and intuitive. Although Ant has an extensive array of options, it also has sensible defaults and good documentation. This open source project is available at http://ant.apache.org/.

Produce clear error messages on invalid configurations. Components must have clear error messages. Nothing is more aggravating than trying to figure out why code you didn't write is degenerating into a `NullPointerException`. Clear error messages can make the difference in a developer's perception of your component.

Provide numerous configuration file examples that are easy to copy. This is especially important if the component is capable of complex configurations. I suggest providing basic examples as well as complex ones. Although it isn't a Java-based component, the best example of this concept I can think of is the command directive file used for the SqlLoader utility that comes with Oracle's database software. In the book *Oracle Database Utilities 10g Release*, chapter 12 gives examples of several database loads. It's easy to find a load directive file to start from that's relatively close to the one you need. This book is available online at http://technet.oracle.com/.

Limit installation and configuration instructions to a one-page cheat sheet. If the instructions are too long and complex, it reduces the benefit of using the component in the first place. Having an expanded document for complete functionality is fine, but users doing basic tasks shouldn't have to read more than a page or two.

Open Source Alternatives

Open source alternatives have blossomed over the past couple of years, so much so that searching for open source alternatives for a given need has become a complicated task. One good place to start is the Open Source Developer's Kit at http://www.osdk.com/. This site lists and categorizes open source components to simplify your task of locating one that fits your need.

Many organizations are wary of using open source products. The perception is that these products are essentially unsupported. And while the price of open source technologies will fit into any corporate budget, many

organizations like the security of having a technical support number to call when problems come up. There are really two issues here. The first is the practical issue of being able to solve technical problems with an open source product to get your applications working, and keep them working. The second is having someone else to blame if your product selection decision turns out to be a bad one. Usually, companies with anti-open source policies are more worried about blame assignment.

Resolving Technical Issues

The following are some steps I've gone through to solve problems with open source component software. Step 1 is the simplest solution. If that doesn't work, you'll need to perform step 2 before trying any of the remaining steps, which are listed in order of simplicity.

Step 1: Search news groups. True, many open source products don't have formal support organizations. But because they are free, open source products tend to have a large user base. Consequently, it's highly likely that someone else has experienced the same problem you're dealing with and consulted a news group about it. Your support mechanism is the Google search engine (http://www.google.com/), which is nice enough to make all the news groups searchable. In fact, I use Google to solve problems with commercial components before calling the support organization because it's usually faster. More often than not, I find the answer in a response to someone else's post.

Step 2: Replicate the problem. This helps you get through the remaining steps quickly. Preferably, you would want to be able to replicate the problem in your development environment. Your test case illustrating the problem should be locally debuggable.

Step 3: Upgrade to the latest version. If a newer version of the product is available, a good time to try it out is when you're trying to resolve a technical issue. Maybe the problem is the result of a bug that someone caught and fixed. Often, the work involved in upgrading is just including a more recent jar(s) in your classpath.

Step 4: Evaluate competing products. I look for competing open source products that do the same thing. If I can easily switch products, I'll try that. I once had an issue with an open source Java FTP client API that had a memory leak. When a news group search didn't reveal an answer to my problem, I switched to another open source FTP client API.

Step 5: Debug your test scenario. Although this isn't a pleasant task, it invariably enables you to determine the root issue. Once you know what the problem is, you can start looking for a way to work around it.

Step 6: Modify component code to fix the problem. This should be treated as an option of last resort. If you use an altered version of an open source product, you'll be on your own for problem investigations after that. You'll never know if the problem you're experiencing was a result of a product bug or the result of your change. If you have to modify the code, take the trouble to report the bug to the developers that produced the component. Most developers of open source products would release a new version with a bug fix. As soon as you can get to an unaltered version of the component, you should do it.

Mitigating Political Risk

The second issue surrounding the use of open source component software is the corporate need to have someone to blame if the choice of open source product doesn't work out as well as intended. I can't think of a way to completely solve the issue, but I can think of many ways to mitigate the risk.

Suggest a commercial software component along with open source alternatives and keep any documentation of the decision. This tactic effectively makes the component decision a group effort. If an open source decision is ever questioned, you can frame the issue as a group decision and point to budgetary advantages.

Track which open source products you use and keep source as well as binary distributions. You won't want to find out at the time you're having trouble that the source for the version you're using is no longer available. As a last resort, you can try to fix the problem yourself.

Identify competitors to the open source product. This shortens the time it takes to switch components should you need to do so.

Component Usage Guidelines

Whether a third-party component is open source or not, one of your goals should be minimizing business risk. A large part of that is keeping switching costs small so that if a mistake is made, it will be easier to correct.

Minimize the classes directly using third-party components. I typically accomplish this by using a proxy pattern. For example, consider a Java FTP

client. I mentioned previously that I switched out this component for one with lower memory requirements. I was able to switch components easily because most of my application did not use the FTP component directly. One class in my application called `FtpClient` performed all FTP requests.

Since `FtpClient` was the only class that used the open source component, I could switch products easily with no impact to the rest of the application. Listing 16.4 shows the source for this client.

Listing 16.4: Sample Component Delegate

```
 1:import com.enterprisedt.net.ftp.*;
 2:
 3:public class FtpClient
 4:{
 5:  public FtpClient ( String hostName,
 6:                     String userName,
 7:                     String password)
 8:         throws ApplicationException
 9:  {
10:    // arg check omitted.
11:    try {
12:       _ftpClient = new FTPClient(hostName);
13:     _ftpClient.login(userName, password);
14:     _ftpClient.setTimeout(DEFAULT_TIMEOUT_IN_MILLIS);
15:     _ftpClient.setConnectMode(FTPConnectMode.ACTIVE);
16:    }
17:    catch (Throwable e)
18:    {
19:      throw new ApplicationException(
20:          "Error creating FTP client. ", e);
21:    }
22:  }
23:
24:  public void put( String localFileName,
25:                   String hostFileName)
26:       throws ApplicationException
27:  {
28:    // arg check omitted
29:    try{_ftpClient.put(localFileName, hostFileName);}
30:    catch (Throwable e)
31:    {
32:      throw new ApplicationException(
33:          "Error with FTP put: local=" + localFileName +
34:          ", remote=" + hostFileName, e);
35:    }
36:  }
37:
```

```
38:  public void get(String hostFileName)
39:       throws ApplicationException
40:  {
41:    // arg check omitted
42:    try{_ftpClient.get(hostFileName);}
43:    catch (Throwable e)
44:    {
45:      throw new ApplicationException(
46:          "Error with FTP get: file=" + hostFileName, e);
47:    }
48:
49:  private FTPClient _ftpClient = null;
50:  }
```

You can use this technique with virtually any software component. Remember that you also must front any component-based exception so that your application relies on internal classes, not external ones.

17

Application Architecture Strategies

To ensure that your applications have internal consistency, you need to establish strategies for logging, exception handling, threading, and configuration management from the outset. Most developers have preferences for each area of application building. If your team has many developers and you let them use their preferences, you will create an application that is internally inconsistent and harder to maintain. This chapter offers strategies for each aspect of application architecture. In this section, I've articulated examples of strategies I've used in the past. Once you establish the strategy, don't be afraid to refine it with additional detail to suit the needs and comfort level of your developers.

Logging Strategies

Application components should not depend on a specific logger. General-use components should be able to use any logging package an application adopts. This can be done by having one class in the application acting as a log manager, delegating the log write to Log4J or one of the other logging packages. Listing 17.1 is an example of an application logger implementation.

Listing 17.1: Sample Log Manager Using Log4J

```
 1:import org.apache.log4j.ConsoleAppender;
 2:import org.apache.log4j.FileAppender;
 3:import org.apache.log4j.PatternLayout;
 4:
 5:public class Logger
 6:{
 7:  public static void logInfo(String message)
 8:  {
 9:    _internalLogger.info(message);
10:  }
11:
12:// Other "log" methods omitted.
13:   private static org.apache.log4j.Logger _internalLogger
14:      = null;
15:   static
16:   {
17:     _internalLogger = org.apache.log4j.Logger.getLogger
18:(Logger.class);
19:     _internalLogger.addAppender (
20:new FileAppender( new PatternLayout(),
21:Environment.getLogFileName()));
22:   }
```

The Apache open source component software, Commons, has a package called Logging designed to fill this need. Unfortunately, it is rather heavy (despite being billed as intentionally "lightweight") and not easy to use. It requires an XML parser and has a more complicated configuration scheme. I consider Logging more hassle than it's worth.

Limit coding for logging methods to one line. Logging is done so frequently that you'll save a lot of time by reducing the code it takes to log. Notice that listing 17.1 makes all logging methods static. You don't have to instantiate anything to log a message. Some logging implementations have you executing a static method on a factory of some type to get a logger. Don't subject your developers to this extra code.

Decouple exception handling and logging. Usually, the activity that follows catching an exception is logging the event with enough information to replicate the bug. Consequently, many developers incorporate logging logic within the exception they generate on construction. This practice is convenient programmatically because the logging takes place automatically when the exception is instantiated, and it ensures that an exception is logged and not swallowed.

However, the strategy creates some problems. It leads to multiple log entries for the same error and log entries without enough context to solve the error (often informational context from the caller is needed). Further, reusing objects in another application that may have a different logging strategy is difficult under this strategy.

Another problem with incorporating logging logic within the exception is that the same exception may be treated as a severe error in one application and as a nonevent in another. For example, "Customer not found on database" might be a grave error worthy of logging for a customer maintenance application but not for a reporting application. It would be logical for the customer look-up to be the same code.

I prefer to log at the deployment level. For example, my enterprise beans log exceptions directly or indirectly generated by classes used by the enterprise beans. Similarly, my client applications and CORBA services log exceptions generated in those deployments. By logging at the deployment level, I can easily have a different logging strategy for each deployment. I can log to a file in a client application deployment and log through JMS for my enterprise bean deployment. This flexibility also allows me to treat exceptions differently in different contexts.

Sample Strategy

- Use `myapp.util.Logger` for all logging. Do not use `System.out.println()`.
- Log errors and exceptions in the deployment and presentation layers as well as in the `run()` method of any asynchronously running code.
- Warnings (i.e., errors not severe enough to warrant the throwing of an exception but would be useful to a developer fixing bugs) can be logged anywhere from any layer.
- When using logging to output debug information, use `logDebug()` so the output can optionally be suppressed.
- Do not use the general logging facility as a transaction log.

Exception-Handling Strategies

Validate all arguments explicitly in all public methods in all public classes. Validating all arguments for publicly consumed methods prevents derivative exceptions. Derivative exceptions are those reported well after the error

occurred. It's an all-too-common error to pass a null argument to a method accidentally, only to find that a `NullPointerException` was generated when the class that was called (directly or indirectly) tries to use the argument and isn't expecting the null value.

While it's possible to get the class, method, and line number where the `NullPointerException` occurred, the actual error occurred well before at the point where the null value originated. This often is in an entirely separate method and class from what's reported in the stack trace and can take significant time to track down, especially if the code is complex and multiple variables could have generated the exception. Typically, derivative exceptions like this take more time and effort to fix because the error message and information don't make the problem clear. Had the method generated an exception message such as "Null name argument not allowed," the error would have been easier to find and fix.

Argument validation enables you to report an error with a clearer message than you would otherwise get. Null arguments are commonly validated. If an argument has to conform to a specific value set (for example, a `docType` argument that allows only pdf, xls, and txt values), the value should be validated as well. I typically use `java.lang.IllegalArgumentException` for invalid arguments, as illustrated in listing 17.2.

Listing 17.2: Example of Argument Checking

```
 1:public static Connection getConnection(
 2:    String connectionPoolName)
 3:     throws NamingException, SQLException
 4:  {
 5:    if (connectionPoolName == null)
 6:       throw new IllegalArgumentException
 7:  ("Null connectionPoolName not allowed.");
 8:    if (connectionPoolName.equals(""))
 9:       throw new IllegalArgumentException
10:  ("Blank connectionPoolName not allowed.");
11:    Context ctx = new InitialContext();
12:    DataSource source = (DataSource)
13:    PortableRemoteObject.narrow
14:                         ( ctx.lookup(connectionPoolName),
15:  DataSource.class);
16:    return source.getConnection();
17:  }
```

Include general catches for public methods in the deployment layer. If you adopt the suggestion of throwing mostly `RuntimeException` exceptions,

your try/catch logic will mainly be in application entry points, such as enterprise beans, servlets, RMI services, and so on. Unless you have a reason to catch errors that are more specific, I recommend a general catch of `java.lang.Throwable`.

This opinion is a bit controversial. Many texts promote the idea that application code should only catch exceptions it is prepared to "handle." I agree with this premise. Most texts go on to posit that an application cannot possibly have a meaningful response to a JVM error such as `OutOfMemoryException` or `ClassNotFoundError`. If you agree with this statement, logic dictates that you should not catch `Throwable`.

However, an application can make a meaningful response to a JVM error. That response typically is to log the error with enough context that a developer can fix the problem. If the application is a Web site or any type of multi-user application, you might also want to notify an operator or application administrator. If you don't catch JVM errors and the like, you leave the reporting of them to chance. While many containers will catch and log `Throwable`, my experience is that those logs are too often ignored.

Many of the applications I support are 24x7x365 with significant business ramifications if they're not functioning. I can't afford to leave the reporting (or notification) for any type of error, JVM or otherwise, to chance. I suspect that many of you can't either.

A common solution is to catch `java.lang.Exception` instead. However, catching `java.lang.Exception` will miss many possible JVM errors (for example, `OutOfMemoryError`). The cost of missing an exception is that a derivative error will probably occur. Catching `java.lang.Throwable` is a better solution because it truly will catch everything, as shown in listing 17.3.

Listing 17.3: Example of Catching Throwable

```
1:try
2:{
3:    …  // App code here
4:}
5:catch (Throwable t)
6:{
7:    _message = t.getClass().getName()
8:        + ":" + t.getMessage();
9:    t.printStackTrace(_log);
10:
11:    _log.println("————————");
12:}
```

Don't catch more specific exceptions unless your error processing is different. In cases where multiple types of checked exceptions can be thrown, I commonly see catches for each type of exception, and the code in the catch block commonly replicated for each exception type. For example, if listing 17.4 contained one catch for `Throwable`, it would have comprised significantly less code.

One word of caution: if your application calls the `stop()` method on a `Thread`, catching `Throwable` will also catch `java.lang.ThreadDeath`. In this case, `ThreadDeath` will have to be rethrown so that the `Thread` actually dies. The `stop()` method was deprecated early on because it's inherently unstable. Use of the `stop()` method is not recommended and actively discouraged. If you don't stop threads with the `stop()` method, there's no reason to worry about accidentally catching `ThreadDeath`.

Listing 17.4: Example of Poor Exception-Catching Code

```
 1:public JMUNonStaticCodeTable
 2:        getAcctConslCodeTable (String aUserId,
 3:            String aSponsorId)
 4:        throws JMUException, Exception
 5:    {
 6:        try
 7:        {
 8:            …… // App code here
 9:        }
10:        catch (JMUException ne)
11:        {
12:            throw ne;
13:        }
14:        catch (Exception e)
15:        {
16:            throw new JMUException(
17:                getJMUTransactionInfo(),
18:                CLASSNAME +
19:                           ".getAcctConslCodeTable" +
20:                "(String aUserId, " +
21:                "String aSponsorId) " +
22:                           "Exception " +
23:                e.getMessage() +
24:                " obtaining accounts/consl for "+
25:                "userId= " + aUserId + " and " +
26:                "sponsorId= " + aSponsorId,
27:                e);
28:
29:        }
```

```
30:         catch (Throwable bigProblem)
31:         {
32:             throw new JMUException(
33:                 getJMUTransactionInfo(),
34:                 CLASSNAME +
35:                             ".getAcctConslCodeTable" +
36:                 "(String aUserId, " +
37:                 "String aSponsorId) " +
38:                 "THROWABLE EXCEPTION OCCURRED "+
39:                 "obtaining accounts/consl for "+
40:                 "userId= " + aUserId + " and "+
41:                 "sponsorId= " + aSponsorId,
42:                 bigProblem);
43:         }
44:         finally
45:         {
46:             return _acctConslCodeTable;
47:         }
48:     }
```

Listing 17.4 illustrates good as well as bad practices. Notice that the programmer did make a deliberate effort to document the context associated with the exception to help a developer debug and fix a problem later. The effort deserves some applause.

Unfortunately, the programmer also put a return statement in the finally block on line 46. This will have the unfortunate effect of swallowing the exception. While the exception contains lots of good information, it will never be seen.

The programmer also had the foresight to catch `Throwable`. I view this as a good point, but some will disagree with me. However, the programmer did miss an opportunity to streamline code. The programmer has much the same processing for the `Exception` catch as the `Throwable` catch. The programmer could have streamlined the code by simply eliminating the `Exception` catch. Listing 17.5 is an alternative version of listing 17.4 that corrects these problems.

Listing 17.5: Improved Exception-Handling Code (Listing 17.4 Rewritten)

```
1:public JMUNonStaticCodeTable
2:     getAcctConslCodeTable (String aUserId,
3:         String aSponsorId)
4:         throws JMUException
5:     {
6:         try
```

```
 7:        {
 8:        ..... // App code here
 9:        }
10:        catch (JMUException ne)
11:        {
12:            throw ne;
13:        }
14:        catch (Throwable bigProblem)
15:        {
16:            throw new JMUException(
17:                getJMUTransactionInfo(),
18:                "getAcctConslCodeTable" +
19:                "(String aUserId, " +
20:                "String aSponsorId) " +
21:                "obtaining accounts/consl for "+
22:                "userId= " + aUserId + " and "+
23:                "sponsorId= " + aSponsorId,
24:                bigProblem);
25:        }
26:
27:    return _acctConslCodeTable;
28:}
```

Make exception messages meaningful. When throwing exceptions, providing null or overly generic messages that state the obvious (for example, "Error occurred") aren't helpful to development staff or end users. Additionally, including the class name in a message is equally meaningless because it's already present in the stack trace. The method name needs to be included only if the method is overloaded. The specific method overload generating the error isn't available explicitly from the stack trace.

Including information about argument values passed can help a developer reproduce the error. However, doing so can be tedious and time consuming if the argument is an object with multiple fields and can also lead to copied code if an object is common and used as an argument to many methods. To make exception handling easier, I make value objects capable of creating textual descriptions of themselves. One way to do this is to implement `Describable` from CementJ. Listing 17.6a illustrates error processing without the benefit of a `Describable` implementation.

Listing 17.6a: Exception Processing Without a Describable Implementation

```
1:public void processOrderWithoutDescribe(
2:    PurchaseOrderDTO order)
3:  {
```

```
 4:    if (order == null)
 5:       throw new IllegalArgumentException
 6:("Null order not allowed.");
 7:    try
 8:    {
 9:      //  App code here
10:    }
11:    catch (Throwable t)
12:    {
13:      StringBuffer errorMessage = new StringBuffer(256);
14:      errorMessage.append("Error processing order: ");
15:
16:      errorMessage.append("custId=");
17:      if (order.getCustomerId() != null)
18:errorMessage.append(order.getCustomerId());
19:      else errorMessage.append("null");
20:
21:      errorMessage.append(", shipDt=");
22:      if (order.getShipDate() != null)
23:errorMessage.append(order.getShipDate());
24:      else errorMessage.append("null");
25:
26:      // Replicate for each field of "order".....
27:
28:      LogManager.getLogger().logError (
29:errorMessage.toString(), t);
30:    }
31:  }
```

Listing 17.6b illustrates how to implement `Describable` to streamline exception processing for every method that uses the `PurchaseOrderVO` object.

Listing 17.6b: Using describe() to Streamline Error Processing

```
 1:public void processOrderWithDescribe(
 2:    PurchaseOrderDTO order)
 3:  {
 4:    if (order == null)
 5:       throw new IllegalArgumentException
 6:("Null order not allowed.");
 7:    try
 8:    {
 9:      //  App code here
10:    }
11:    catch (Throwable t)
12:    {
13:      LogManager.getLogger().logError (
```

```
14:"Error processing order: " + order.describe(), t);
15:     }
16:   }
```

Use native JDK exceptions before creating your own. There's no need to reinvent the wheel. Many developers won't even check if there is an appropriate JDK-defined exception that they could use in lieu of creating an application-specific exception. As a real-world example, I've seen a developer create an exception called `NullValueException` that was thrown when a method was provided a null argument instead of a valid value. `IllegalArgumentException` (from `java.lang`) would have been a better choice.

Exploit Java's unchecked exception capability. Methods that throw `RuntimeException` don't force all callers into coding a try/catch block or listing the exception in the throws clause and will reduce the size and complexity of the caller. Callers still have the option of trapping exceptions with a try/catch or listing the exception in the throws clause, but they are not forced to. In addition, most developers create checked exceptions out of habit, not because of a deliberate choice. I've been guilty of this, too.

My suggestion to use unchecked exceptions instead of checked exceptions is a bit unorthodox and controversial. I used to use checked exceptions religiously. After coding thousands of try/catch blocks, I realized that using `RuntimeException` does save tremendous amounts of code and makes the remaining code much more readable. The response to most exceptions (whether they are checked or unchecked) in most applications is to log the error with enough context that it can be duplicated and fixed by a developer later. Using `RuntimeException` allows you to choose where to place your try/catch code instead of forcing it to be a part of most methods in most classes. In many cases, the cost of throwing checked exceptions (in terms of extra coding/maintenance time) is not worth the benefits.

Use of the `RuntimeException` is appropriate. According to the JavaDoc for the JDK, `RuntimeException` is intended for "exceptions that can be thrown during the normal operation of the Java Virtual Machine." Most application-level exceptions fall into this category. I use `RuntimeException` in most of the applications I write and place the try/catch blocks in methods that have the ability to record enough context that I can replicate and fix the error.

I'm not alone in my opinion. Johnson (2002) provides a good argument for using unchecked exceptions instead of exceptions. Bruce Eckel (of "Thinking in Java" fame) also appears to have converted to using unchecked exceptions

(see http://www.mindview.net/Etc/Discussions/CheckedExceptions) along with Gunjan Doshi (see http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html).

Limit the nesting depth of a try/catch block to two. (Many of you, I know, would campaign for one.) If you need more, remove the inner blocks of code to separate private methods for readability. In addition, fixing bugs in nested try/catch scenarios can be difficult. As an aside, the need for deeply nested try/catch logic usually indicates a need to refactor this section of code.

Don't catch exceptions and do nothing with them. For programmatic convenience, some developers catch exceptions but fail to code the catch block. This practice eliminates a compiler error but makes the code harder to maintain. Many times, swallowing exceptions leads to derivative exceptions later on that are harder to find and fix. If you catch an exception, do something with it (at least log it).

Never put a return statement in a finally block. If something throws an exception within a try block, the finally block is executed before the exception is thrown. If you issue a return statement within the finally or something excepts within the finally, the original exception will never be thrown and never be seen. This will increase the time and effort needed to debug a problem.

The architect and project manager should establish an exception-handling and logging strategy before coding begins. Developers often have personal preferences for exception handling and logging. If you don't define a strategy for exception handling and logging, developers will choose their own, and you'll have no consistency across the application. Eventually, when different sections of the application are integrated, conflicts will arise. In addition, it'll be more difficult for an outsider to maintain the application.

For example, suppose one developer adopts the philosophy of logging exceptions when they're instantiated, while another expects logging to occur at the deployment level. When all code is integrated, some errors will go unreported, which will greatly increase testing and maintenance time.

One of the most valuable pieces of information generated by an exception is the stack trace of the root exception. The stack trace indicates where the exception was thrown definitively. In some cases, you will even get an exact line number. Ideally, you should see the stack trace of the root exception

combined with more descriptive context information. To accomplish this, I've incorporated two exceptions that are smart enough to record information from the root exception in CementJ: `ApplicationException` and `ApplicationRuntimeException` from the `org.cementj.base` package.

An important component of your exception-handling strategy should be a "root" exception that is the parent of all non-JDK-related exceptions generated by your application. `ApplicationRuntimeException` or `ApplicationException` would be wise choices.

You also need to enforce your exception-handling strategy. I use group code reviews as a mechanism for enforcement and education. Code reviews, if conducted properly, are constructive. Developers can learn a lot by reviewing the code of other developers. Reviews also make it more difficult to skirt established policy, such as an exception-handling strategy. Additionally, code reviews allow you to identify any shortcomings in the exception-handling strategy and make adjustments, if necessary.

Sample Strategy

- Use `IllegalArgumentException` to flag all erroneous method arguments on public methods.
- Always include enough information in the message of the exception to duplicate the condition in a testing environment.
- All application exceptions should extend `ApplicationRuntimeException` (from `org.cementj.base`). New application exception proposals should be reviewed by the technical architect.
- All try/catch blocks in the business logic layer or the data access layer should not interfere with the throwing of an `ApplicationRuntimeException`. Throw the exception to the caller instead.

Threading Strategies

Most applications have tasks that run asynchronously. These tasks can include scheduled "batch" work and long-running tasks users won't wait for. J2EE applications are no exception. Unfortunately, the J2EE specification makes no provision for work done asynchronously.

Furthermore, the J2EE specification explicitly prohibits spawning threads directly or indirectly from within enterprise beans, so if you're going to

adhere to the specification, your hands are tied. This is one of the few negative features of the J2EE specification. It should offer a way to spawn asynchronous tasks with the understanding that the container could not provide transactional integrity for spawned tasks (e.g., you couldn't roll them back).

Many developers provide a message-driven bean deployment for their asynchronous work so they can effectively spawn the task via JMS. Rather than an intentional misuse of JMS, this practice is a way to spawn asynchronous work while complying with the J2EE specification. It has the effect of twisting the container's arm into handling the threading issue.

You have several choices for spawning asynchronous work in J2EE:

- ▲ Delegate the work out to a message-driven bean.
- ▲ Delegate the work out to an RMI service that spawns needed threads.
- ▲ Break the specification and spawn threads in stateless session beans.
- ▲ Use the Timer service that's part of the EJB 2.1 specification (if you are running a 2.1-compliant container).

I favor delegating the work out to an RMI service. RMI services have no threading restriction, and their calls offer a tighter coupling than JMS message transmission. And because most container vendors offer clusterable RMI services, you're not sacrificing availability or scalability by using an RMI service to do asynchronous work.

Although it's more programmatically convenient to break the specification, it's also risky. With many containers, you can get away with it in a stateless session bean context, but your code may not work in some containers and may break on a container upgrade.

Only intermediate to advanced developers should attempt to write thread-safe code. Bugs from threaded code can be the most difficult bugs to find and fix.

The following are just a few recommendations for creating thread-safe code. It is by no means a comprehensive review of threading. Entire books have been written on multithreaded and concurrent programming. My favorite is Lea (2000).

Limit thread count to a manageable level. Each thread consumes memory and causes more context switching within the JVM. Because of the resources and management overhead required, adding threads has diminishing returns. The point at which returns for additional threads diminish varies

among hardware configurations. Essentially, more threads are not necessarily better and do not necessarily increase throughput. In fact, depending on your hardware, more threads could decrease throughput.

Avoid explicitly setting thread priority. Only an advanced developer should consider explicitly setting the priority of threads because it greatly increases the complexity of debugging for a multithreaded application because it increases the chance of thread starvation. Thread starvation most often happens when a thread doesn't get execution time because the JVM is occupied with higher-priority tasks. The symptom of thread starvation is that work you spawned doesn't appear to ever run. Unless you have a specific reason for wanting a thread to have a lower priority, you should avoid the additional complexity.

Setting thread priority would be especially dangerous if you're breaking the J2EE specification and initiating multithreaded code from within enterprise beans. You could inadvertently interfere with the normal operation of a container.

Identify all threads as daemon threads. The JVM makes a distinction between daemon and user threads. The JVM shuts down when all user threads terminate. If you spawn a user thread, it's best to make a provision for how it terminates. If you declare your threads as daemon threads, they do not interfere with a JVM shutdown. By the way, user threads are the default. Keep in mind that anything running daemon threads can be abruptly terminated at any point. Listing 17.7 is one way to declare daemon threads.

Listing 17.7: Declaring Threads as Daemons

```
1:// app code here
2:
3:Thread thread = new Thread(runnableClass);
4:thread.setDaemon(true);
5:thread.start();
6:
7:// app code here
```

Log enough information on error so that you can reproduce an error situation in a single-threaded test case. Debugging multithreaded code is often more difficult. Although most IDEs do support multithreading, the behavior of threads under an IDE often differs from their behavior at normal runtime. Error processing in spawned tasks needs to be robust enough to allow debugging a section of code within a single-threaded test case.

Explicitly name all created threads. Many containers have management extensions that will let you view the activity of each thread. As containers usually have dozens, if not hundreds, of threads, a specific thread is easier to find if it is descriptively named. Listing 17.8 illustrates how to name a thread.

Listing 17.8: Naming a Thread

```
1:// app code here
2:
3:Thread thread = new Thread(runnableClass);
4:thread.setName("My thread name");
5:thread.start();
6:
7:// app code here
```

Perform a code review for all multithreaded code. I generally promote code reviews for the entire business logic and data access layers. Since multithreaded code can be especially difficult to debug, you should consider a code review even if you don't subject any of the other layers to the same level of scrutiny.

Until recently, programmers were left to their own devices to code, execute, and manage multithreaded code. On a recent project, I had an extensive need for multithreaded code, but the skill level of the developers wasn't high enough to get everything coded safely within the project timeline. To make the deadline, I created a package to manage execution of our threaded code. I then recrafted and re-architected this package into the open source product called ThreadWorks (available at http://sourceforge.net/projects/threadworks/).

Sample Threading Guidelines

- ▲ Code all asynchronous tasks to implement `java.lang.Runnable`.
- ▲ Run all asynchronous tasks using ThreadWorks.
- ▲ Perform a code review of all asynchronous tasks.
- ▲ Centrally define all `TaskManagers`. Consult the technical architect for a `TaskManager` assignment for all asynchronous tasks.

Configuration Management Strategies

Most J2EE applications have some configurable properties, such as:

- ▲ Names for database connection pools

▲ Logging level indicators

▲ Mail group names for error messages

I usually have one class that's responsible for reading and interpreting configuration files, and I put static accessors on that class to make the properties available. An example appears in listing 17.9.

Listing 17.9: Implementing an Environment Class

```
1:public class SampleEnvironment
2:{
3:    // Some code omitted
4:    public static String getDatabaseConnectionPoolName()
5:    {
6:        return _myEnvironment.getProperty("db.pool");
7:    }
8:}
```

Having each object read the configuration file and search for the properties they care about is more correct from an object-oriented design perspective. However, it simply isn't practical in many cases. If your application has 300 classes that need access to application properties, having them all read a configuration file is a bit disk intensive. Many times, the number of needed properties exceeds what can practically be fed into the JVM as a system property (with the "-D" option) at runtime.

The advantage of using one class is that configuration file management occurs in one place in the application. Developers can find and change the class easily, and it's simple to reference the application properties from other classes. You can even change configuration file formats and not affect the rest of your application.

CementJ provides a base class called `ApplicationEnvironment` (from package `org.cementj.base`) that provides basic functionality. `ApplicationEnvironment`, by default, provides support for configuration files in a properties format (see `java.util.Properties`), but it's possible to override that and use XML or other formats. `ApplicationEnvironment` also checks for file updates at a configurable interval. This feature is useful because it allows you to change the behavior of your application without stopping and restarting your EJB container.

Listing 17.10 is an example implementation of `ApplicationEnvironment`. See CementJ JavaDoc for additional details.

Listing 17.10: Implementing the ApplicationEnvironment Class

```
 1:package book.sample.env;
 2:
 3:import org.cementj.base.ApplicationEnvironment;
 4:
 5:public class SampleEnvironment
 6:    extends ApplicationEnvironment
 7:{
 8:    protected SampleEnvironment(){}
 9:
10:    protected String getConfigurationFileName()
11:        {return CONFIG_FILE_NAME;}
12:    private static final SampleEnvironment
13:        _myEnvironment = new SampleEnvironment();
14:
15:    public static String getDatabaseConnectionPoolName()
16:    {
17:        return _myEnvironment.getProperty("db.pool");
18:    }
19:
20:    public static final String     CONFIG_FILE_NAME
21:                                 = "myapp.properties";
22:}
```

Another example of establishing strategies for exception handling, logging, and coding conventions drawn from the open source community can be found at http://jakarta.apache.org/cactus/participating/coding_conventions.html.

This example is the coding conventions established for the open source product Cactus that is used to facilitate writing test cases for server-side classes, such as servlets or enterprise beans. The URL above is an excellent example of formally establishing architectural policies discussed in this chapter and communicating them. The exception-handling and logging strategies do differ somewhat from the advice I've provided here. I still provide it as an example because establishing clear guidelines for developers to follow is more important than the minor disagreements I have with some of the line items.

Further Reading

Johnson, Rod. 2002. *Expert One-on-One: J2EE Design and Development*. Indianapolis, IN: Wrox Press.

Lea, Doug. 2000. *Concurrent Programming in Java Second Edition: Design Principles and Patterns*. Boston, MA: Addison-Wesley.

Section 4

Testing and Maintaining J2EE Applications

Once the work of building the application is finished, the technical architect is often asked to lead performance-testing activities and ensure that the application is production ready. At this stage, the architect's primary objective is improving application performance, stability, and operational readiness. To achieve this goal, the architect needs to conduct performance tests and make performance improvements, recommend changes to make applications easier to monitor and support, and identify candidates for code refactoring.

This section guides you through these activities. In it, you will learn how to:

- ▲ Establish coding guidelines for functional test cases.
- ▲ Conduct effective performance tests.
- ▲ Effectively profile your application to improve its use of memory and CPU.
- ▲ Improve supportability for your applications.
- ▲ Recognize when code needs refactoring.
- ▲ Effectively apply code-refactoring techniques.

18

Functional Testing Guidelines

Throughout the book, I've recommended that you create test cases for all data access objects and business logic objects because these layers contain most of the application's complexity. I've also mentioned that these tests are used for unit testing as well as part of a regression test suite. In this chapter, I show you how to use open source testing components, based on JUnit, to accomplish this.

JUnit is an open source framework that facilitates the writing and running of test cases and grouping them into test suites. You can download JUnit from the project home page at http://www.junit.org/index.htm.

I use open source testing components because they're popular and easy for anyone to access. Those of you using commercial testing packages can look at the examples in this chapter as conceptual. All commercial testing tools I'm aware of will support the testing concepts illustrated in this chapter. Performance and load testing concepts are covered in chapter 19.

Ideally, I'd recommend highly automated ways to test the presentation layer. Currently, the most popular product to do this appears to be Apache's Cactus. However, this product has limited testing capabilities. For instance, although it can test if session attributes are set properly, it can't tell if the aesthetics of the resulting page are working. Consequently, you have to do manual testing before release anyway, which significantly reduces the benefits of setting up a regression test for the presentation layer.

Additionally, the setup work for Cactus is verbose and tedious. Between the two drawbacks I've mentioned, the cost of setting a regression test up with Cactus usually outweighs the benefits. I'm sure that this will change given time. You can download Cactus from the project home page at http://jakarta.apache.org/cactus/.

Testing Assumptions

I subscribe to two beliefs about functional testing:

- ▲ Automated testing is better than manual testing.
- ▲ Finding bugs sooner lowers costs and improves quality.

Automated testing is better because it's consistent and easier to run. It doesn't slack off at the end of a hard day, and it's easily repeatable. As a result, automated regression testing for even small changes in the application is more cost-effective. I've seen projects that go so far as to incorporate the test suite in the build script. If one of the regression tests fails, the build fails and the developers are forced to fix the bug.

Automated testing is as complete as you want it to be. If you subscribe to the view that test cases should be created as the application is developed, then you also feel that developers should initially create test cases. But these test cases are usually not complete. If you adopt the recommendation that the first step in fixing a bug is writing a test case that reproduces it, your regression test is enhanced as you find more bugs. Over time, you get a robust set of regression tests that increases the probability of finding bugs before deployment.

Automated testing isn't free. Creating and maintaining tests consumes manpower, so it's important to use the 80/20 rule. Initially code test cases for the 80 percent of the application that is most complex and most prone to error. You can fill out the remaining 20 percent over time. If you stop to create automated tests that are 100 percent comprehensive, you'll end up adding many hours to the timeline to the extent that the costs will outweigh the benefits.

The sooner you find a bug, the less damage it will cause. Damage from bugs occurs in many forms. A bug discovered late in the testing process can impact a project timeline. A bug discovered after deployment can damage users' faith in the system and the development team. And fixing bugs late in the process involves more manpower. At that point, the fix can involve not

only the developer but an entire testing team (if your project has them, and many do).

By contrast, news of a bug caught early in the project passes no further than the development team. The testing team and end users won't be directly involved in verifying a fix to a bug they never experienced.

As applications mature and gain complexity, automated testing becomes preventative medicine. With complex code comes the likelihood that a developer will fix one problem and inadvertently cause another. Some would say that this is a red flag indicating a need for code refactoring, as discussed in chapter 20. Automated testing decreases the possibility that a fix to one problem causes another and then goes unnoticed until after production.

Testing Coverage

Most developers already code test cases for classes they write to allow for unit testing and debugging. Given this, writing formal test cases that can later be executed in a test suite usually doesn't take much additional work. This section shows you how to write test cases within the JUnit framework.

Selectively choose test case coverage. Ideally, you should code test cases for all public methods in all classes. Realistically, however, most projects achieve something less than the ideal, at least in the beginning. Again, I initially apply the 80/20 rule. It's more important to have basic test cases for the most complex places in the application because these places have a greater chance of containing bugs. It's most convenient to create test cases as you write the code. I typically ask developers for one test case per public method for every data access object and business object in the application.

Asking for one test case per method or service call is by no means comprehensive, but it's a sensible start. Over time, this test case library can be enhanced and will provide good automatic regression testing capabilities. If developers use a testing framework such as JUnit, the test cases should be combinable into test suites.

When investigating a bug report, most developers will try to replicate the bug. At some point in the investigation, the developer will find out which classes contain the bug. For most bugs, it's not too much additional work to create a test case (or modify an existing one) to replicate the bug and include it in the test suite. This gives the developer a smaller section of code to debug. It also contributes to a more robust regression test. The regression test should be executed on a periodic basis (even on a scheduled basis, with errors mailed to the development team).

Test Case Coding Overview and Examples

The mechanics of setting up a JUnit test case are relatively straightforward. All test cases extend `junit.framework.TestCase`. You need to override the `runTest()` method, which performs pretest processing, the test itself, and posttest processing. I usually code a `main()` so the test case can be run and debugged outside the test suite. Listing 18.1 is an example of a test case from a ProjectTrak data access object.

Listing 18.1: Sample Test Case for a Data Access Object

```
 1:package test.dvt.app.project.dao;
 2:
 3:import com.dvt.app.project.dao.ProjectTaskDAO;
 4:import com.dvt.app.project.vo.ProjectTaskVO;
 5:
 6:import org.cementj.util.DatabaseUtility;
 7:import org.cementj.base.DataNotFoundException;
 8:import junit.framework.TestCase;
 9:
10:import java.sql.Connection;
11:
12:public class TestProjectTaskDAO extends TestCase
13:{
14:
15:  public TestProjectTaskDAO()
16:  {
17:      super("ProjectTaskDAO unit tests");
18:  }
19:
20:  protected void setUp() throws java.lang.Exception
21:  {
22:    super.setUp();
23:    _dbConnection =
24:        DatabaseUtility.getOracleJDBCConnection
25:            (    "localhost", 1521, "ORA92",
26:                "scott", "tiger");
27:  }
28:
29:  protected void runTest() throws java.lang.Throwable
30:  {
31:      try
32:      {
33:        this.setUp();
34:        ProjectTaskDAO dao =
35:            new ProjectTaskDAO(_dbConnection);
36:
```

```
37:         ProjectTaskVO taskVO = null;
38:
39:         // Test for data not found.
40:         boolean dataNotFound = false;
41:         try      {taskVO = dao.getProjectTask(77777);}
42:         catch    (DataNotFoundException d)
43:         {
44:             dataNotFound = true;
45:         }
46:         TestCase.assertTrue("Test 1: Not found test",
47:                             dataNotFound);
48:
49:         // test for data found
50:         taskVO = dao.getProjectTask(11111);
51:         TestCase.assertTrue("Test 2: Select test",
52:                             taskVO != null);
53:
54:         // test for task save
55:         taskVO.setTaskId(77777);
56:         dao.saveProjectTask(taskVO);
57:         ProjectTaskVO newTaskVO =
58:             dao.getProjectTask(77777);
59:         TestCase.assertTrue("Test 3: Insert test",
60:                         newTaskVO.equals(taskVO));
61:
62:         // test for delete
63:         dao.deleteProjectTask(77777);
64:         dataNotFound = false;
65:         try      {taskVO = dao.getProjectTask(77777);}
66:         catch    (DataNotFoundException d)
67:         {
68:             dataNotFound = true;
69:         }
70:         TestCase.assertTrue("Test 4: Delete test",
71:                             dataNotFound);
72:       }
73:       finally
74:       {
75:           this.tearDown();
76:       }
77:   }
78:
79:   protected void tearDown() throws java.lang.Exception
80:   {
81:     super.tearDown();
82:     DatabaseUtility.close(_dbConnection);
83:   }
84:
85:   private Connection         _dbConnection = null;
```

```
86:
87:  public static void main(String[] args)
88:  {
89:    TestProjectTaskDAO test = new TestProjectTaskDAO();
90:
91:    try
92:    {
93:      test.runTest();
94:    }
95:    catch (Throwable t)      {t.printStackTrace();}
96:  }
97:}
```

I usually put the database connection creation in the setup() method override and the close in the teardown() method override.

Here are a few things to note from the example in listing 18.1:

- ▲ The test is self-contained for the most part.
- ▲ All assertions and the test case are uniquely labeled so developers can find them easily if they fail in the test suite.
- ▲ The test case is in a package that only contains test cases and supporting classes.
- ▲ The test case is named so it's easy for other developers to find.

Combining Test Cases into Suites

I've mentioned that it's fairly easy to combine `TestCase` classes into test suites. JUnit implements this capability via its `TestSuite` class. Once it's instantiated and loaded with `TestCase`, the `TestSuite` class can be run via JUnit's `TestRunner` class, which is a GUI utility. Listing 18.2 is a sample test suite from one of my open source products.

Listing 18.2: Sample Test Suite from ThreadWorks

```
1:public class ThreadworksTestSuite {
2:
3:  public ThreadworksTestSuite() {}
4:
5:  public static Test suite()
6:  {
7:    TestSuite suite= new TestSuite("Threadworks");
8:
9:    suite.addTest(new InstantiationTest() );
10:   suite.addTest(new SingleTaskWithoutNotification() );
11:   suite.addTest(new SingleTaskWithNotification() );
```

```
12:    suite.addTest(new TerminationTest() );
13:    suite.addTest(new GroupTaskWithNotification() );
14:    suite.addTest(new GroupTaskWithoutNotification() );
15:    suite.addTest(
16:    new TaskCollectionWithNotification() );
17:    suite.addTest(
18:    new TaskCollectionWithoutNotification() );
19:    suite.addTest(new SuccessorTasksWithNotification() );
20:    suite.addTest(
21:    new ScheduleNonRecurringWithoutDependencies() );
22:    suite.addTest(
23:    new ScheduleNonRecurringWithDependencies() );
24:
25:    return suite;
26: }
27:
28:  public static void main(String[] argv)
29:  {
30:    junit.swingui.TestRunner.run(
31:    ThreadworksTestSuite.class);
32:
33:  }
34:}
```

Testing Best Practices

Keep test cases and supporting classes in a separate package structure. Although test cases are developed in conjunction with the application, they are not truly a part of it. Typically, you have no need to deploy them to anything but your testing environments.

I usually organize the package structure of my test cases and supporting classes after the application package structure. For instance, for CementJ, the test classes for everything in package `org.cementj.base` are in `test.cementj.base`. Supporting classes for those test cases are in package `test.cementj.base.support`. Keeping a consistent package structure will save developers time.

Adopt a naming convention for test classes that makes them easy to find. This is another suggestion that can save developers time. For instance, I name all of my test classes `Test`*`Xxx`*, where *`Xxx`* is the name of the class being tested. For example, class `TestValueObject` is the test class for `ValueObject`. I prefer to combine all unit tests for the same class into one test class, but this is not a technical requirement or suggestion.

Put a descriptive label on all test assertions. When you code `assert()` or `assertTrue()` method calls in your test cases, you can optionally provide a label that will be displayed on failure; for example:

```
TestCase.assertTrue("Test 2: Select test", taskVO != null);
```

This label becomes important when you're running the test case as part of a regression test suite. If the test fails, you don't want developers wasting time trying to figure out which test failed. Note that the error description doesn't need to identify the test class that failed; JUnit will do that for you.

I've adopted the practice of prefixing all assertion descriptions with `Test` *x*`:`, where *x* is the test number within the test case. I make *x* unique within the test class. This is boring and uncreative, but functional.

Make each `TestCase` class self-sufficient and independent. Test cases should not rely on other test cases having to execute before it. If you write the test cases so that test case 1 has to be run before test case 2 will work, it's not obvious to other developers what the prerequisites are should they want to use a test case for unit testing and debugging. There may be some isolated cases where implementing this suggestion isn't practical.

19

Performance Tuning and Load Testing

Management often looks to the technical architect to lead performance-tuning efforts. This chapter provides tips and tricks you can use to tune and load test your J2EE applications. In fact, you can use many of the concepts presented here for other types of applications as well.

Establish performance and scalability targets. This is the first step in performance tuning and load testing. Without targets, you'll never know when those tasks have finished.

Although there are always opportunities for performance improvement, performance tuning has diminishing returns over time. When you start tuning, the changes you make will result in larger performance improvements. But over time, your improvements will get smaller and smaller with most applications. Because most of the benefit you get from tuning will occur in the first 20 percent of the work, the 80/20 rule applies once again.

Don't start tuning until after the application is in testing. Many developers have a desire to tune every piece of code they write. I admire their desire for completeness, but it hurts the timeline of the project. Chances are high that a good percentage of the code being tuned at this level will not result in good performance enhancement to the application as a whole. Although I've meet many developers that are not comfortable with this concept,

experience has taught me that at some places in the application, the cost of tuning doesn't reap enough benefit for anyone to care about.

Most performance problems originate in application code. Developers tend to ferret out performance problems by examining container configurations, JVM options, operating system performance, network performance, and the like rather than looking at code. This is usually wishful thinking on the part of developers.

Measure performance before tuning to establish a baseline. The next section discusses how to measure performance. The numbers that result from performance measuring will be the basis for judging the effectiveness of performance improvements.

Performance tuning is an extensive subject. I can only scratch the surface here. Joines, Willenborg, and Hygh (2002) is a good comprehensive reference for performance tuning J2EE applications. I find the numerous tips in Bulka (2000) incredibly useful for improving performance after I've identified a problem.

Measuring Performance

A load generator is a software package that measures the performance of applications, including J2EE applications. Typically, a load generator mimics multiple users so you can simulate load.

Most load generators operate by running a test script (or a set of them) several times concurrently, simulating a configurable number of virtual users. By measuring the performance each virtual user gets, the load generator enables you to examine the performance averages over all virtual users. Sometimes this information is presented graphically.

Load tests are usually written in the form of URL sequences. I prefer not to set up tests for other classes unless the application has significant back-end processing that needs to be tested under load. This can happen if your application processes JMS messages from other applications, for example.

Although a load generator can tell you *if* you're meeting performance targets, it can't tell you *why* your performance is what it is. If you meet your performance targets, you should stop the tuning effort. If you don't meet your targets, you'll need to apply additional techniques to diagnose the causes of the performance problem.

Use load tests to detect memory leaks in addition to performance. Yes, even though Java has a garbage collector for memory, it's still possible to

have a leak. It's fairly easy to determine if you have a memory leak; it's much harder to find where it is.

Memory Leaks Defined

With Java, memory leaks occur in code that retains references to objects that are no longer needed. A **reference** is a variable declaration and assignment. Java's garbage collector periodically frees memory associated with nonreferenceable variables. If a variable is referenceable, its memory will not be freed.

For instance, the variable `account` in the following code is a reference for a value object:

```
AccountVO account = new AccountVO();
```

If this line of code appears as a local variable declaration within a method, the reference ends when the method completes. After the method completes, the garbage collector frees memory associated with the `account` declaration.

If the declaration is an instance-level field, the reference ends when the enclosing object is no longer referenced. For example, if the variable `account` is declared as an instance-level field for `CustomerVO`, the reference to account ends when the reference to an instantiated `CustomerVO` object ends, as shown here:

```
public class CustomerVO
{
  private AccountVO account = new AccountVO();
}
```

A variable defined as `static` can easily cause a memory leak because the reference ends when the JVM stops or the reference is specifically nulled out.

Memory leaks in J2EE applications are frequently caused by statically defined `Collection` objects. For instance, it's common to statically define an application `Properties` object to store configuration details, as in the following:

```
public class Environment
{
    private static Properties _configurationProps =
new Properties();
}
```

Any value stored in this statically defined `Properties` object is referenceable. `Collection` objects, such as `Properties` or `HashMaps`, often produce memory leaks because it's easy to put values into them and forget to remove them later.

Testing for Memory Leaks

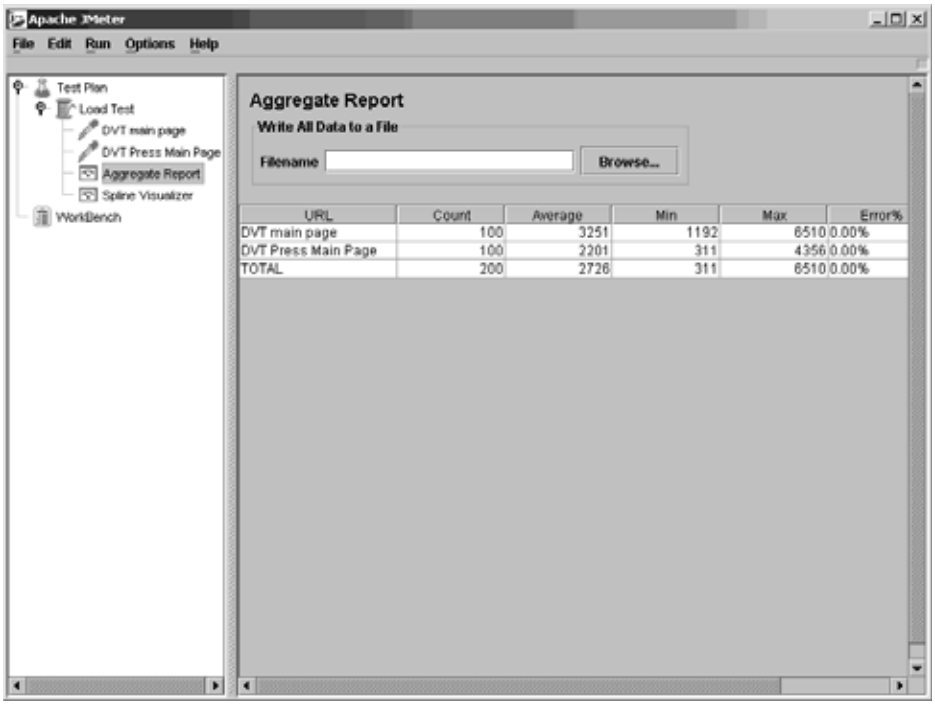
To test for memory leaks, note how much memory the container is using before, during, and after the test. You should see memory at a low level to start, ramp up during the test, and slowly decrease within a few minutes to an hour after the test. For example, a container might initially start at 128MB of memory and grow to 180MB during the performance test. After the test, memory allocation should trend back toward 128MB.

It's not realistic for memory to return to its pretest level, but most of the memory allocated during the test should be freed. To diagnose memory leaks, I start by running the test in a profiler with the memory options turned on. The next section tells you how to do this.

One output of memory leak testing is knowing which transactions are not performing to requirements. In the layered application architecture discussed in this book, the business logic layer or the data access layer are the most likely places for performance problems to occur. Beans and Web services usually only publish functionality in business objects anyway. It should be relatively quick to construct a test case (or modify an existing one) specifically for the underlying business objects that produce the performance problem. Constructing these test cases gives you something that's easier to profile.

I use an open source load generator from Apache called JMeter. It's easy to install and set up test plans. You can obtain JMeter at http://jakarta.apache.org/jmeter/. A JMeter test plan can contain a thread group with any number of URLs. You dictate how many "virtual users" get created to run the test plan and how long they run. If you set up a URL for each major page and control in your application, you'll get average, minimum, and maximum timing information for each URL, as illustrated in figure 19.1. JMeter allows you to save this information in a separate file for future reference later.

Figure 19.1: JMeter Example



Always run load tests under the same conditions multiple times. You need to make sure that nothing is interfering with the test that you're not aware of. I've seen load tests accidentally contending for batch runs or server backups. If you run each test at least twice and get similar results each time, then you'll have a higher degree of trust in the information JMeter gives you.

Document and limit changes between each test. If you make several changes between load tests, you won't know which of the changes helped or hurt performance. For example, let's say you changed four things. It's possible that one of the changes helped performance, one of the changes hurt performance, and two didn't make any material difference whatsoever. Because you combined the changes, however, you'll never really know what helped performance and what hurt it.

Monitor and record CPU and memory usage during the test. The simplest way to do this is at the operating system level. Most UNIX operating systems provide a `top` utility, which provides CPU and memory usage for each

process as well as usage for the entire server. You're obviously interested in what's happening for the container process during the load test. Listing 19.1 is an extract from a `top` utility output. If your application runs on a Windows platform, you'll need to use the `perfmon` utility.

Listing 19.1: Sample Top Utility Output

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
21886 dashmore  15   0  1012 1012   776 R     0.1  0.1   0:00 java
1 root       15   0   480  480   428 S     0.0  0.0   0:04 init
2 root       15   0     0    0     0 SW    0.0  0.0   0:00 keventd
3 root       15   0     0    0     0 SW    0.0  0.0   0:00 kapmd
```

It's more convenient to limit `top` utility output to the process running the container. Unfortunately, the options to the `top` utility are different for each platform. For instance, on Solaris, using the `-U` option will limit output to a specific user.

```
top -U username
```

On Linux, you would limit output by specifying the process ID with the `-p` option.

```
top -p pid
```

If you're using a UNIX other than Solaris or Linux, you'll have to consult the manual page for the `top` utility for your specific UNIX platform. By the way, Loukides (2002) is an excellent reference for interpreting CPU and memory utilization statistics that UNIX utilities provide.

Expect to see both CPU and memory usage increase during the test and decrease after the test. If you don't see memory allocation diminish after the test (e.g., within fifteen to thirty minutes), it's likely that you have a memory leak. You should profile your load test both for CPU usage and memory usage.

Diagnosing Performance Problems

Use profiler tools to diagnose performance problems. A profiler reports the activity of a JVM at a configurable interval (typically, every five milliseconds) and reports the call stack in use for every thread. The methods taking the most time will most likely show up in more observations and provide leads as to where you should tune.

Some J2EE containers use multiple JVMs, making it difficult to profile the entire container. Instead, you'll want to directly profile test cases that use the underlying business objects. You'll skip profiling the deployment

and presentation layers in their entirety, but performance tuning is most likely to be at the business logic layer or lower anyway. In a layered architecture, the deployment and presentation layers don't perform much of the processing. If your container only uses one JVM, you can profile the entire container and run your JMeter test script against it with a small load.

Do not attempt to profile in a clustered environment. For those of you in a clustered architecture, I recommend profiling in one instance only, not in clustered mode. Your goal is to tune your application, not wade through the work your container does to implement clustering.

Profilers tell you where (in which class or method) CPU time is being spent and where (in which classes) memory is being allocated. The default profiler that comes with the JVM (HPROF) produces output that's not intuitive and is hard to read, but that output contains much of the same information as commercial profilers. The advantage of commercial profilers is that they make performance information easier to read and interpret. If your organization has a license for a commercial profiler, use it instead of HPROF.

If you don't have access to a commercial profiler, you'll probably have to spend a few more minutes interpreting the output than your colleagues with commercial profilers. In the next section, I provide a usage cheat sheet for HPROF. The default profiler measures both CPU time and memory allocation, but I recommend measuring these separately to avoid contaminating the test. Methods for debugging memory leaks are also included in the next section.

Using HPROF to Measure CPU Usage

HPROF is invoked by including arguments when the JVM is started. To measure CPU usage, include the following arguments in the Java command invocation:

```
-Xrunhprof:cpu=samples,thread=y,file=cpu.hprof.txt,depth=32
```

HPROF will place results in the file listed in the `file` argument. The `cpu` argument indicates that HPROF will measure CPU consumption, not memory. The `thread` argument tells you that HPROF will indicate the thread in the stack trace details. And the `depth` argument indicates how many levels or calls to record in the stack trace details.

The JVM should be shut down cleanly for HPROF to have an opportunity to record its observations. Don't be surprised if HPROF takes a few minutes to record its data. Likewise, don't be alarmed at the sluggishness of your application while HPROF is running; that's to be expected.

With the CPU options turned on, HPROF produces a file. The first place to look is in the last part of the file, in a section detailing CPU stack trace rankings. This section provides a call stack ID and the percentage of the time the call stack was invoked. HPROF works by recording the call stacks for each thread every five minutes. Odds are high that HPROF will most frequently observe the places where your application is spending the most time. Listing 19.2 illustrates the CPU stack trace rankings produced by HPROF.

Listing 19.2: Sample CPU Stack Trace Rankings

```
CPU SAMPLES BEGIN (total = 52) Sun Jul 13 10:48:18 2003
rank   self  accum   count trace method
   1 30.77% 30.77%      16     6 java.lang.StringBuffer.<init>
   2 25.00% 55.77%      13     5 java.lang.StringBuffer.<init>
   3  9.62% 65.38%       5    11 java.lang.Class.getName
   4  7.69% 73.08%       4     7 java.lang.Class.isAssignableFrom
   5  7.69% 80.77%       4     8 java.lang.Class.getName
   6  5.77% 86.54%       3    10 java.lang.Class.isAssignableFrom
   7  3.85% 90.38%       2    13 java.lang.reflect.Field.copy
   8  1.92% 92.31%       1    12 java.lang.Class.isAssignableFrom
   9  1.92% 94.23%       1     4 java.io.FileOutputStream.writeBytes
  10  1.92% 96.15%       1     1 sun.misc.URLClassPath$3.run
  11  1.92% 98.08%       1     9 java.lang.Class.getName
  12  1.92% 100.00%      1    14 java.lang.Class.copyFields
CPU SAMPLES END
```

Once you have the call stack ID, you can get details of what's in that stack in the preceding section of the HPROF-produced file. Listing 19.3 shows the stack corresponding to trace 6, which accounted for 30.77% of the CPU time.

Listing 19.3: Stack Trace Details

```
TRACE 6: (thread=3)
        java.lang.StringBuffer.<init>(StringBuffer.java:115)

        org.cementj.base.ValueObject.getConcantonatedObjectValue(ValueObject.java:219)
        org.cementj.base.ValueObject.equals(ValueObject.java:49)

book.sample.dto.cementj.TestCustomerDTO.main(TestCustomerDTO.java:28)
```

The stack trace description will tell you what class or method in your code is using up the time. You want to look at the first class in the trace that

is a part of your application (the bold line in listing 19.3). This is where you need look for tuning opportunities.

Listing 19.4 is the section of code highlighted in the trace. Lines 3 and 4 in the listing are taking the largest amount of CPU time. The only piece of this that can be tuned is the initial size of the buffer, given by `_startBufferSize`. A higher number means that it will take longer to instantiate the buffer, but the `append()` operations later in this method won't take as long because the memory is already allocated.

Listing 19.4: Extract from CementJ

```
1:  private String getConcantonatedObjectValue()
2:  {
3:    StringBuffer buffer =
4:        new StringBuffer(_startBufferSize);
5:    Object tempObj = null;
6:    Object[] tempArray = null;
7:    //  Some code omitted.
8:  }
```

I've found that taking more time in the initial allocation is a better practice than causing the `append()` to reallocate larger and larger chunks of memory. After validating that the `_startBufferSize` is being estimated appropriately and isn't much larger than the memory needed, there isn't any way to tune this code. I would move on to the other hot spots listed in the trace.

Entire books have been written about coding for better performance. The first book I consult is Bulka (2000), which has a wide range of code-level tuning suggestions that are supported by performance test data. I can't recommend this book highly enough.

Look only at stacks using 5 percent or more of the CPU. The rest is too small to worry about. Suppose you're able to tune a method only using 1 percent of your CPU. Suppose you get a 20 percent performance improvement for that method. Since it only uses 1 percent of the CPU anyway, your effort will improve performance by just 0.2 percent overall—usually not considered a material improvement. The corollary to this suggestion is that if all stacks are using less than 5 percent of your CPU, you can stop tuning.

If you find that most of your time is being spent in JDBC, you should enlist the aid of a database administrator and tune your database and/or application SQL. It's entirely possible that the stack trace indicates a specific method in a DAO. That usually will limit the SQL being executed to one or two statements. You can then execute these statements via an online query

tool to diagnose query performance. Within the online query tool, you can try out alternative ways of writing the query to get your sample working faster.

Using HPROF to Measure Memory Usage

To have HPROF measure memory usage, invoke the JVM with the following arguments:

```
-Xrunhprof:heap=sites,file=mem.hprof.txt,depth=24,cutoff=0.01
```

Memory leaks are among the most difficult bugs to find and diagnose. The largest objects that can cause memory leaks are statically defined variables, particularly statically defined collections. HPROF provides a trace ranking that details the traces that have the most memory allocated. Listing 19.5 is a sample of a memory trace ranking.

Listing 19.5: Sample Memory Trace Ranking

```
SITES BEGIN (ordered by live bytes) Sun Jul 13 11:12:39 2003
          percent          live       alloc'ed  stack class
 rank   self  accum    bytes objs   bytes objs trace name
    1 35.21% 35.21%   158256 3297 33600000 700000   419
       java.lang.reflect.Field
    2 17.42% 52.64%    78312  492   79680  517     1 [C
    3 10.10% 62.74%    45416  214   45416  214     1 [B
    4  5.03% 67.77%    22608  471 4800000 100000   415
       book.sample.dto.cementj.CustomerDTO
    5  4.19% 71.96%    18840  471 4000000 100000   418
       java.lang.reflect.Field
    6  3.91% 75.87%    17576  315   17576  315     1 java.lang.Object
    7  3.35% 79.22%    15064  282   15064  282     1 [S
    8  2.98% 82.20%    13384  239   13384  239     1 java.lang.Class
    9  2.89% 85.09%    12984  541   13632  568     1 java.lang.String
   10  2.48% 87.58%    11168  208  601448  220     1 [I
SITES END
```

The first aspect of the memory ranking to look at is the ratio of "live" bytes to "allocated" bytes. Live bytes represent memory that is referenceable and currently being used by the application. Allocated bytes represent the total memory allocated within the JVM. The difference between the allocated and live bytes will be garbage collected at some point.

Memory leaks tend to be traces where the live bytes represent close to 100 percent of the allocated bytes. The first trace in listing 19.5 doesn't fit this profile, but the second one does. It's important to remember that a high

live-to-allocated bytes ratio could indicate a leak, but it's possible to have a high ratio without a leak.

Take a closer look at trace 419, shown in listing 19.6. As the largest memory consumer, trace 419 might be a good place to find something you can tune. The next largest memory consumer, trace 1, might be a place to see a memory leak.

Although you might not recognize the object type consuming the memory, it's important to look at the stack trace because it indicates the code that's allocating the memory. The trace behind trace 419, which is allocating about 35.21 percent of live memory, is presented in listing 19.6.

Listing 19.6: Trace 419 from the Sample in Listing 19.5

```
TRACE 419:
        java.lang.reflect.Field.copy(Field.java:83)
        java.lang.reflect.ReflectAccess.copyField(ReflectAccess.java:9)
        sun.reflect.ReflectionFactory.copyField(ReflectionFactory.java:
277)
        java.lang.Class.copyFields(Class.java:1962)
        java.lang.Class.getDeclaredFields(Class.java:1090)
        org.cementj.base.ValueObject.<init>(ValueObject.java:26)
        book.sample.dto.cementj.CustomerDTO.<init>(CustomerDTO.java:8)
        book.sample.dto.cementj.TestCustomerDTO.main(
TestCustomerDTO.java:38)
```

Once again, look at the first object in the trace that comes from the application. It appears that memory is being allocated from within the constructor of `ValueObject` on line 26. In listing 19.7, line 3 shows the code in `ValueObject` allocating the memory that is highlighted in the trace.

Listing 19.7: Code Highlighted in the Trace

```
 1:protected ValueObject()
 2:  {
 3:    _classField = this.getClass().getDeclaredFields();
 4:    for (int i = 0 ; _classField != null &&
 5:                    i < _classField.length; i++)
 6:    {
 7:      _classField[i].setAccessible(true);
 8:    }
 9:    _startBufferSize = (_classField.length + 1) * 128;
10:  }
```

Unfortunately, little can be done to reduce memory allocation here. `ValueObject` needs access to the underlying field definitions to properly

implement meaningful versions of `describe()`, `equals()`, `hashcode()`, and several other methods.

The next trace in the ranking, trace 1, is shown in listing 19.8.

Listing 19.8: Trace 1 from the Sample in Listing 19.5

```
TRACE 1:
        <empty>
```

It turns out that trace 1 doesn't have a lot of meaningful information. I see this occasionally in HPROF traces but have never run across an explanation as to why it happens. My guess is that it's either an HPROF bug, memory associated with the JVM internally, or memory that no longer has a reference and is awaiting garbage collection. In practice, I'd move on to the next item in the ranking that doesn't belong to trace 1.

Further Reading

Bulka, Dov. 2000. *Java™ Performance and Scalability*. Vol. 1, *Server-Side Programming Techniques*. Reading, MA: Addison-Wesley.

Joines, Stacy, Ruth Willenborg, and Ken Hygh. 2002. *Performance Analysis for Java Websites*. Reading, MA: Addison-Wesley.

Loukides, Mike, and Gian-Paolo Musumeci. 2002. *System Performance Tuning*, 2nd ed. Sebastapol, CA: O'Reilly & Associates.

20

Postimplementation Activities

The project isn't over when it's over. Most development teams spend time after implementation to correct bugs and bring the product additional stability. In some organizations, separate teams provide postimplementation support. Members of the support teams receive training from the development team.

The technical architect can play a significant role in providing stability to the implemented application and reducing the time and effort needed to maintain it. In addition, by performing a constructive review of the development process, both its successes and mishaps, the architect can help prevent problems on future projects. In many organizations, the architect is responsible for providing support in addition to developing applications.

This chapter presents guidelines to help you improve the quality, completeness, and timeliness of the information you gather on application issues and problems. In addition, I describe techniques for responding more quickly and effectively to your application monitoring information, including tips on debugging and refactoring. The postimplementation activities you learn here can help you decrease the quantity and severity of problems over time and minimize the number of users experiencing outages.

Application-Monitoring Guidelines

Many organizations consider application monitoring as a simple matter of purchasing the right software—BMC Patrol™ or EcoTools™, for example. Usually, however these tools are not used effectively. And even when used to their potential, they monitor server and database health much better than they do custom applications.

Monitoring software can determine if your application is available and measure the performance of common tasks but not much more. No product currently on the market can tell you *why* your application isn't available or find bugs that prevent users from getting what they want. Although purchasing a monitoring tool may be the first step in effective application monitoring, the task certainly doesn't end there.

The objectives of monitoring applications are to detect and notify application administrators of ongoing issues and problems. You can address these objectives in several ways without using monitoring software. Here are some guidelines for using your application's functionality to its best advantage.

Look for every opportunity to improve the quality and completeness of your regression test. The first place to discover application issues and problems is in the application regression test. In earlier chapters, I recommended constructing test cases along with the application. Now you need to combine these test cases into a regression test. Bugs and errors caught at this stage are often found and fixed without incident. The more comprehensive your regression test is, the greater your chances of catching bugs before your application is released.

Improve the quality and completeness of application error reports. The best sources of information you have about application issues and problems after code has deployed are the error reports your application generates (reports of runtime errors). Application runtime errors are usually either software bugs or the result of environmental issues (e.g., the database is unavailable). With environmental issues, the earlier you act on the error report, the fewer users are affected.

Using the information in the application error report, you can solve most bugs resulting in log entries or reports. I still recommend writing test cases that replicate bugs and including them in your regression test later. But any time you can save identifying the problem will in turn save time for your support staff.

Broadcast application error messages to provide opportunities for quicker responses. Often an environmental issue shows up as a system error report and can be corrected before too many users are affected. I have found that most developers look at logs only when an end user has reported a problem. Therefore, I usually broadcast error reports to the development team over some medium (e.g., e-mail) to increase the probability that someone will fix the errors before users notice.

Some developers don't like receiving broadcasted application error reports because the volume of broadcast reports can be substantial in the beginning. Usually, however, the volume of error reports decreases over time as application bugs are fixed.

Make the first sentence or subject of an error report descriptive and meaningful. Because applications commonly generate many reports of the same error, you can save the support staff time by providing a reasonable description of the error in the subject line of an e-mail or in the first sentence of the message. This is especially important in companies in which the support staff does not develop applications.

System error reports tend to be more comprehensive than user bug reports. In my experience, users don't report all the errors they experience. And often the information a user provides is less than complete.

Nonetheless, user bug reports can be valuable sources of information. Often users can help you replicate the bug. Because the goal of using test cases and system error reports is to reduce the number of bugs reported by end users, the quantity of bugs reported is an indicator of the quality of your test cases and error reports.

Bug-Fighting Guidelines

Always fix root errors first. I distinguish between root and derivative errors. A derivative error happens because of an unexpected condition caused by some other error. That other error is the root error. If you fix the root error, the derivative error often disappears without your having to fix it directly.

For example, I once received reports that an application was exceeding the maximum number of database connections with one of the connection pools. It turned out that this error was derivative. The real problem was that locking contention with a table made a transaction much longer than it should have been. As a result, the application was holding on to database connections longer than anticipated. Fixing the locking problem made the connection

pool errors disappear. Many would have been tempted to increase the connection pool size.

Include tests for bugs in the test suite. As stated earlier, making the regression test more robust enables you to identify problems earlier. XP advocates would use stronger terms and advise testing for all bugs in the test suite. The reality is that some bugs, like those that cause display defects, are difficult to code into a test case. Sometimes the cost of producing the test far outweighs its benefits. Support staff will have to evaluate the need for a bug-inspired test on a case-by-case basis.

Declare war on derivative error. A `NullPointerException` is a good example of a derivative error—a problem that is reported well after it occurred. As discussed in chapter 17, you can reduce derivative exceptions by checking the validity of argument values in public methods.

Continually refine error reports to be more useful and descriptive. If you receive an error report that doesn't provide enough information to solve the problem, you have two bugs: the one causing the error report and the other causing the incomplete report. It's usually worthwhile to fix both problems. If you continually enhance the information in error reports, you will notice a decrease in the amount of time it takes to investigate bugs and diagnose problems.

Top Refactoring Indicators

Refactoring is rewriting selected code to make it easier to understand and maintain. Fowler (2000) provides an extensive list of conditions that indicate the need to refactor—conditions he calls "Bad Smells in Code." Although his list is so comprehensive I wouldn't presume to add to it, it is code centric. For readers who may not have the intimate understanding of code needed to apply Fowler's advice, I describe some observable symptoms that may indicate a need to refactor but don't require a full audit of the application's source.

Classes that you can't change without inadvertently creating other bugs may need to be refactored. This symptom is reminiscent of the movie *Night of the Living Dead.* Some programming bugs don't die, they just come back in different forms. Various circumstances can cause a bug to undergo such a metamorphosis.

Sometimes this happens when code within a class behaves differently

depending on context. For example, I had one client that used a central API to provide reports for multiple applications. For political reasons, the API interpreted some of the argument values differently depending on which application was calling it (not a good idea, I know). Eventually, this service needed to be refactored because we couldn't change it without inadvertently causing bugs in some of the applications calling it.

Sometimes bugs morph when code within a single class is doing too much and should be separated into multiple classes. For example, one application I worked on had to be able to accept data from multiple data sources. Some of the data sources were relational databases; some weren't. At first we had only two data sources. The programmer took a shortcut and put conditional logic in the class managing input to handle either data source. When we had to add data sources, the class had to be refactored.

Enhancements or bug fixes requiring identical changes in multiple classes may indicate a need to refactor. Some developers are almost too fond of copy-and-paste technology. In most cases, identical code in multiple classes should become common code "called" by multiple classes. Given a tight time frame, the developer who discovers a case of copied code might not have the time to make the code common. The architect or manager can assist by providing a mechanism to track these cases so they can be fixed when time permits.

Abnormally complicated methods or classes that developers fear changing may need to be refactored. Sometimes this symptom occurs in combination with the morphing-bug symptom described earlier. It is another indication that the code is too complex and needs to be refactored. Of course, the validity of this symptom depends on the assumption that developers are rational and their "fear" justified, which might not always be the case.

Common Refactoring Techniques

Refactory has been the topic of entire books, within which are dozens of refactoring techniques. Here I concentrate on the most commonly used techniques. Readers interested in delving deeper into the topic should see Fowler (2000) and Alur et al. (2003).

Extract and Delegate

Common code identified in multiple classes calls for the extract-and-delegate method of refactoring. The common code is extracted and placed in a

separate class. The new class then serves as a delegate and is called within any class needing it. Many of the static utilities in CementJ were created because common code existed in many classes and needed to be centralized.

For example, it is common for JDBC-related classes to close `ResultSets`, `Statements`, and `PreparedStatements` in a finally block. Unfortunately, closing one of these objects can throw an `SQLException`, which is a checked exception. As shown in lines 9 through 18 of listing 20.1, this causes nested try/catch logic in the finally block, which is usually identical everywhere.

Listing 20.1: Sample Candidate for the Extract-and-Delegate Method of Refactoring

```
 1:PreparedStatement pStmt = null;
 2:try
 3:{
 4:      pStmt = connection.prepareStatement(sqlText);
 5:    // JDBC Code here
 6:}
 7:finally
 8:{
 9:    if (pStmt != null)
10:    {
11:        try {pStmt.close()}
12:        catch (SQLException q)
13:        {
14:             Logger.logWarning(
15:                "Error closing PreparedStatement",
16:                 q)
17:        }
18:    }
19:}
```

Although the nested try/catch isn't complicated, it's verbose and makes JDBC code harder to read. Listing 20.2 illustrates how extracting that code into a separate utility class shortens the code quite a bit.

Listing 20.2: Using the Extract-and-Delegate Technique (Refactoring of Listing 20.1)

```
1:PreparedStatement pStmt = null;
2:try
3:{
4:      pStmt = connection.prepareStatement(sqlText);
5:    // JDBC Code here
6:}
7:finally
```

```
 8:{
 9:    DatabaseUtility.close(pStmt);
10:}
```

One argument I usually get from the copy-and-paste advocates is that the central utility created by the extract-and-delegate technique is more complex than the original. This can be true, but the result is much less code to maintain. Further, a central utility is usually the most tested because it's used most often. In listing 20.2, for example, the source for the utility method isn't much more complex than the original. Listing 20.3 provides the source for `DatabaseUtility.close()`.

Listing 20.3: DatabaseUtility Source Extract

```
 1:  public static void close(PreparedStatement pStmt)
 2:  {
 3:    if (pStmt == null) return;
 4:    try  {pStmt.close();}
 5:    catch (SQLException e)
 6:    {
 7:       LogManager.getLogger().logWarning(
 8:                "Prepared statement close error", e);
 9:    }
10:  }
```

Common code isn't always a static utility but comprises common characteristics shared among multiple classes. To centralize this code, you would use the extract-and-extend technique.

Extract and Extend

This refactoring technique is used when classes share the same characteristics and methods. Ideally, you will identify such classes in design, but sometimes the commonality becomes apparent only after construction. `ValueObject` from CementJ is a good example of applying the extract-and-extend technique. In fact, I created `ValueObject` after noticing a lot of common code with value object classes for one of my applications.

Often value object classes need functions such as producing a textual description of themselves or implementing `hashcode()` and `equals()` so they can be effectively used in hash constructs. `ValueObject` makes these kinds of functions common so they do not have to be coded in multiple value object classes individually. To benefit from `ValueObject`, all you need do is extend it. `ValueObject` is covered extensively in chapter 10.

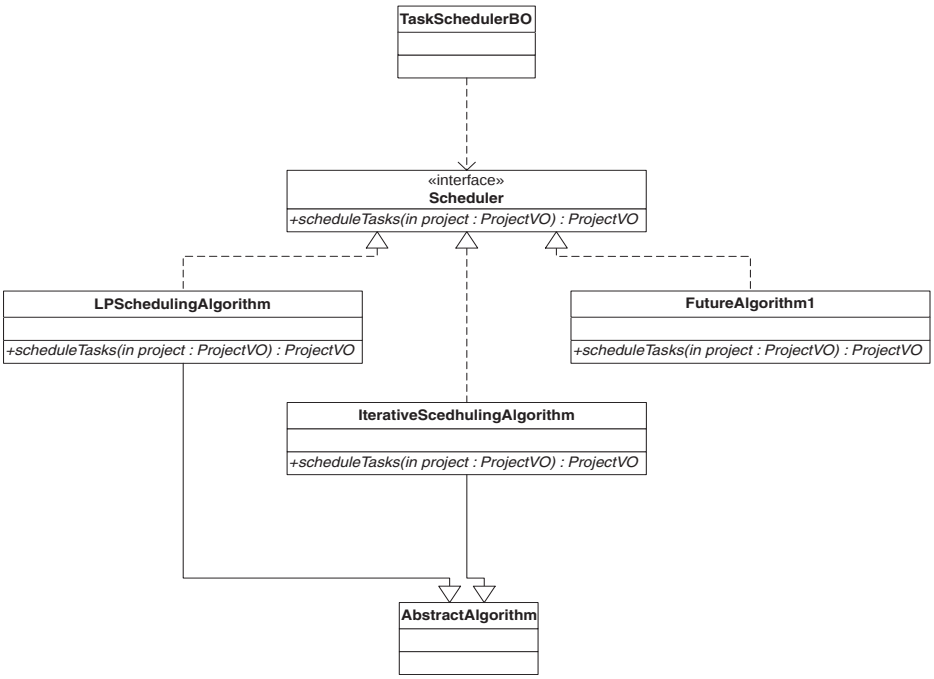
A combination of the extract-and-delegate and extract-and-extend techniques is also commonly used.

Extract and Decouple with Interface

You use this refactoring technique when classes in your application have the same function but do not have common code. This technique is a combination of the two techniques previously described.

One example can be drawn from ProjectTrak and its ability to use multiple scheduling algorithms. As discussed in chapter 13, ProjectTrak users can specify the algorithm used to produce a project schedule to facilitate testing and refinement of the scheduling feature. Some of these algorithms may have common characteristics and share common code, but others may not. Not all scheduling algorithms can usefully extend `AbstractAlgorithm`. By decoupling with the interface `Scheduler` as illustrated in figure 20.1, the user can implement any algorithm without adversely affecting other parts of the application.

Figure 20.1: Sample Interface Decoupling



Further Reading

Alur, Deepak, John Crupi, and Dan Malks. 2003. *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. New York: Prentice Hall.

Fowler, Martin. 2000. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Bibliography

Alur, Deepak, John Crupi, and Dan Malks. 2003. *Core J2EE patterns: Best Practices and Design Strategies*, 2nd ed. New York, Prentice Hall.

Beck, Kent. 2000. *Extreme Programming Explained.* Reading, MA: Addison-Wesley.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1998. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

Bradley, Neil. 2000. *The XSL Companion*. Reading, MA: Addison-Wesley.

Brooks, Frederick P., Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Bulka, Dov. 2000. *Java™ Performance and Scalability.* Vol. 1, *Server-Side Programming Techniques*. Reading, MA: Addison-Wesley.

Castro, Elizabeth. 2002. *HTML for the World Wide Web with XHTML and CSS: Visual QuickStart Guide*, 5th ed. Berkeley, CA: Peachpit Press.

Cockburn, Alistair. 2001. *Writing Effective Use Cases*. Boston: Addison-Wesley.

Cohen, Frank. 2003 (March). "Discover SOAP Encoding's Impact on Web Service Performance." *IBM DeveloperWorks*. Available online at http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/.

Date, C. J. 2003. *An Introduction to Database Systems*, 8th ed. Boston: Pearson Addison-Wesley.

DeMarco, Tom, and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams*, 2nd ed. New York: Dorset House.

Ensor, Dave, and Ian Stevenson. 1997. *Oracle Design*. Sebastopol, CA: O'Reilly & Associates.

Fleming, Candace C., and Barbara von Halle. 1989. *Handbook of Relational Database Design*. Reading, MA: Addison-Wesley.

Fowler, Martin. 2000. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

———. 2003. *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley.

Fowler, Martin, and Kendall Scott. 1997. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Grand, Mark. 1998. *Patterns in Java*, Vol. 1. New York: John Wiley & Sons.

———. 1999. *Patterns in Java*, Vol. 2. New York: John Wiley & Sons.

———. 2002. *Java Enterprise Design Patterns*. New York: John Wiley & Sons.

Goldratt, Eliyahu. 1992. *The Goal: A Process of Ongoing Improvement*. Great Barrington, MA: North River Press.

———. 1994. *It's Not Luck*. Great Barrington, MA: North River Press.

———. 1997. *Critical Chain*. Great Barrington, MA: North River Press.

Hall, Marty. 2000. *Core Servlets and JavaServer Pages (JSP)*. New York: Prentice Hall.

Horstmann, Cay S., and Gary Cornell. 2001. *Core Java 2*. Vol. 2, *Advanced Features*, 5th ed. Essex, UK: Pearson Higher Education.

Hunt, Craig. 2002. *TCP/IP Network Administration*, 3rd ed. Sebastopol, CA: O'Reilly & Associates.

Hunter, Jason, and William Crawford. 2001. *Java Servlet Programming*, 2nd ed. Sebastopol, CA: O'Reilly & Associates.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson. 2001. *Extreme Programming Installed*. Reading, MA: Addison-Wesley.

Johnson, Rod. 2002. *Expert One-on-One: J2EE Design and Development*. Indianapolis, IN: Wrox Press.

Joines, Stacy, Ruth Willenborg, and Ken Hygh. 2002. *Performance Analysis for Java Websites*. Reading, MA: Addison-Wesley.

Kroll, Per, and Philippe Krutchen. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Boston: Addison-Wesley.

Lea, Doug. 2000. *Concurrent Programming in Java Second Edition: Design Principles and Patterns.* Boston, MA: Addison-Wesley.

Loukides, Mike, and Gian-Paolo Musumeci. 2002. *System Performance Tuning*, 2nd ed. Sebastopol, CA: O'Reilly & Associates.

McConnell, Steve. 1998. *Software Project Survival Guide*. Redmond, WA: Microsoft Press.

Richardson, Chris. 2003. "Simplifying Domain Model Persistence in a J2EE Application Using JDO." *TheServerSide*. Available at http://www.theserverside.com/resources/article.jsp?l=JDODomainModel.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.

Spielman, Sue. 2003. *The Struts Framework: Practical Guide for Java Programmers*. Boston: Morgan Kaufmann.

Sun Microsystems. 2002. *Java™ 2 Platform, Enterprise Edition (J2EE™) Specification ("Specification") Version: 1.4*. Mountain View, CA: Sun Microsystems.

Taylor, David. 1990. *Object-Oriented Technology: A Manager's Guide*. Reading, MA: Addison-Wesley.

The Apache Software License, Version 1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)."

   Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.
4. The names "Apache" and "Apache Software Foundation" and "Apache JMeter" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.
5. Products derived from this software may not be called "Apache", "Apache JMeter," nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Index

Design and Build J2EE Applications On-Time and On-Budget

This handbook is a concise guide to architecting, designing, and building J2EE applications. It guides technical architects through the entire J2EE project, including identifying business requirements, performing use-case analysis, doing object and data modeling, and leading a development team through construction. Whether you are about to architect your first J2EE application or are looking for ways to keep your projects on-time and on-budget, you will refer to this handbook again and again.

You will discover how to:

- Design robust, extensible, and easy to maintain J2EE applications.
- Apply commonly used design patterns effectively.
- Identify and address application architecture issues *before* they hinder the development team.
- Avoid common mistakes that derail project budgets and timelines.
- Guide the development team through the design and construction process.
- Set up effective procedures and guidelines that increase stability and decrease error reports.
- Document and communicate the application design to target the development team's work.
- Identify and address application architectural issues.
- Effectively estimate needed resources and timelines.

"Derek Ashmore has assembled a 'must have' book for anyone working with Java and/or J2EE applications." — Dan Hotka, Author/Instructor

"This book is very well crafted and explains everything you really need to know in order to be a successful and productive J2EE architect." — Ian Ellis, Senior Technical Architect

"It is concise, to the point, and packed with real-world code examples that reinforce each concept." — Ross MacCharles

Derek Ashmore is a technical architect and consultant with more than fifteen years of experience in a wide range of technologies, including Java and J2EE. He frequently architects and designs high-usage Web applications and manages development teams to build them. Many of his articles have been published in the *Java Developer's Journal, JavaPro,* and other trade publications.
Derek Ashmore can be reached at dashmore@dvt.com.

ISBN 0-9729548-9-9
53495>
9 780972 954891

DVT Press