Making Everything Easier!™

Pattern-Oriented Software Architecture

FOR DUMMIES®

Learn to:

- Understand software architecture basics and implement best practices
- Recognize and utilize patterns, layers, pipes, filters, and MVC
- Create patterns and build your own pattern collection
- Incorporate plans for emerging platforms and technologies into your projects

Robert Hanmer

Get More and Do More at Dummies.com®

Start with **FREE** Cheat Sheets

Cheat Sheets include

- Checklists
- Charts
- Common Instructions
- And Other Good Stuff!

To access the Cheat Sheet created specifically for this book, go to
www.dummies.com/cheatsheet/patternorientedsoftwarearchitecture

Get Smart at Dummies.com

Dummies.com makes your life easier with 1,000s of answers on everything from removing wallpaper to using the latest version of Windows.

Check out our

- Videos
- Illustrated Articles
- Step-by-Step Instructions

Plus, each month you can win valuable prizes by entering our Dummies.com sweepstakes. *

Want a weekly dose of Dummies? Sign up for Newsletters on

- Digital Photography
- Microsoft Windows & Office
- Personal Finance & Investing
- Health & Wellness
- Computing, iPods & Cell Phones
- eBay
- Internet
- Food, Home & Garden

Find out "HOW" at Dummies.com

**Sweepstakes not currently available in all countries; visit Dummies.com for official rules.*

Pattern-Oriented Software Architecture

FOR

DUMMIES®

Pattern-Oriented Software Architecture FOR DUMMIES®

by Robert Hanmer

WILEY
A John Wiley and Sons, Ltd, Publication

Pattern-Oriented Software Architecture For Dummies®
Published by
John Wiley & Sons, Ltd.
The Atrium
Southern Gate
Chichester
West Sussex
PO19 8SQ
England

Email (for orders and customer service enquires): cs-books@wiley.co.uk

Visit our home page on www.wiley.com

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

For technical support, please visit www.wiley.com/techsupport.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats. For more information about Wiley products, visit us at www.wiley.com.

British Library Cataloguing in Publication Data: A catalogue record for this book is available from the British Library.

ISBN 978-1-119-96399-8 (pbk); ISBN 978-1-119-96631-9 (ebk); ISBN 978-1-119-96632-6 (ebk); ISBN 978-1-119-96630-2 (ebk)

Printed and bound in the United States by Bind-Rite

10 9 8 7 6 5 4 3 2 1

About the Author

Robert Hanmer is a director of The Hillside Group, an organization whose mission is to improve quality of life for everyone who uses, builds, and encounters software systems. The Hillside Group also sponsors Pattern Languages of Programming (PLoP) software pattern conferences. Bob is active in the software pattern community and has been program chair at pattern conferences in the United States and overseas.

He is a consulting member of technical staff with Alcatel-Lucent near Chicago. Within Alcatel-Lucent, Lucent Technologies, and Bell Laboratories (same office, new company names), he is involved in development and architecture of embedded systems, focusing especially on the areas of reliability and performance. Previously, he designed interactive graphics systems used by medical researchers.

Bob is the author of *Patterns for Fault Tolerant Software* (Wiley) and has written or co-written 14 journal articles and several book chapters. He is a senior member of the Association for Computing Machinery, a member of the Alcatel-Lucent Technical Academy, and a member of the IEEE Computer Society. He received his BS and MS degrees in Computer Science from Northwestern University in Evanston, Illinois.

Dedication

For Karen

Author's Acknowledgments

First, and most important, I want to acknowledge the authors of *Pattern-Oriented Software Architecture: A System of Patterns* (Wiley): Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Peter also has been helpful with questions about modern C++ and the software architecture classroom.

Many other people answered questions, reviewed sections, or generally consulted with me while I was writing this book. Thanks to Ademar Aguiar, Omar Aldawud, Dan Bergen, Filipe Correia, Chuck Corwin, Jerry Dzeidzic, Christoph Fehling, Becky Fletcher, Brian Foote, Karen Hanmer, Kenji Hiranabe, Lise Hvatum, Satomi Joba, Dr. Ralph Johnson, Capt. U.S. Navy (Ret.) Will H. Jordan, Steven P. Karas, Allan Kelley, Christian Kohls, Christian Koppe, John Krallman, John Letourneau, Steffen Macke, Dennis Mancl, Jyothish Maniyath, Veena Mendiratta, Pedro Monteiro, Karl Rehmer, Linda Rising, Hans Rudin, Eugene Wallingford, Michael Weiss, and Joe Yoder.

Thanks to the members of my writers' workshop group at PLoP 2011 who held a workshop on parts of this book: Dr. Tanya L. Crenshaw, Andre Hauge, Jiwon Kim, Alexander Nowak, Rick Rodin, YoungSu Son, and Hironori Washizaki.

The Real-World Example sidebars in the pattern chapters are based on a workshop at the 1998 OOPSLA conference. It was organized by Michael Duell, Linda Rising, Peter Sommerlad, and Michael Stal. Russ Frame, Kandi Frasier, Rik Smoody, and Jun'ichi Suzuki participated in the workshop and contributed to the examples that I've adapted here.

Thanks also to the many people at John Wiley & Sons, including Birgit Gruber, Chris Katsaropoulos, Elizabeth Kuball, Ellie Scott, Jim Siddle, Kathy Simpson, Chris Webb, and the others whose names you see on the Publisher's Acknowledgments page.

Publisher's Acknowledgments

We're proud of this book; please send us your comments at http://dummies.custhelp.com. For other comments, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

Some of the people who helped bring this book to market include the following:

Acquisitions and Editorial

Project Editor: Elizabeth Kuball

Executive Commissioning Editor: Birgit Gruber

Assistant Editor: Ellie Scott

Copy Editor: Elizabeth Kuball

Technical Editor: James Siddle

Editorial Manager: Jodi Jensen

Sr. Project Editor: Sara Shlaer

Editorial Assistant: Leslie Saxman

Cover Photo: © teekid / iStock

Cartoons: Rich Tennant (www.the5thwave.com)

Composition Services

Senior Project Coordinator: Kristie Rees

Layout and Graphics: Joyce Haughey

Proofreaders: John Greenough, Tricia Liebig

Indexer: Sharon Shock

Marketing

Associate Marketing Director: Louise Breinholt

Marketing Manager: Lorna Mein

Senior Marketing Executive: Kate Parrett

Marketing Assistant: Tash Lee

UK Tech Publishing

Michelle Leete, Vice President Consumer and Technology Publishing Director

Martin Tribe, Associate Director–Book Content Management

Chris Webb, Associate Publisher

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Consumer Dummies

Kathleen Nebenhaus, Vice President and Executive Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Contents at a Glance

Table of Contents

Introduction

Wouldn't it be great to never rewrite code? To always face new challenges rather than solve the same problems over and over? To always solve new and interesting problems instead of rehashing old ones? If you remember how you solved a problem before, reuse that solution. Don't reinvent the wheel!

Software patterns help you avoid reinventing the wheel, in that they help you avoid reinventing the solution to a software problem that someone else has already solved.

Patterns have been around in the software community since at least the early 1990s. Software pattern authors have been writing patterns that document their proven solutions in the hope that you — the reader — will benefit from their experience.

In particular, many people are collecting and publishing patterns that structure software architecture — the underlying structure of the software. The goal of architectural patterns is to speed your development; allow you to move forward, knowing that a particular architecture will help rather than hinder you; and ultimately give you the time you need to solve new and interesting problems.

Pattern-Oriented Software Architecture For Dummies is written to help you understand the basics of software architecture. It also helps you understand software patterns. The book brings these two concepts together and presents eight software architectures that you can use in your next software design project. It also gives you some design patterns, tips, and resources where you can find out more about software patterns.

About This Book

This book provides proven architectures and designs expressed as patterns. These patterns aren't the only ways you can structure your software architecture, though, and this book doesn't replace the other references you use for software design patterns.

As you read this book, keep in mind that you can't just plug-and-play these patterns. *Your* intelligence and taste are required to adapt these patterns to *your* design problem. This is the norm with software patterns: No respectable pattern author will tell you that you can use his or her patterns without adapting them to your situation.

In the early days, software patterns provided valuable assistance to people who were trying to get a handle on object-oriented design. The discussions of these patterns seemed to me, however, to focus on getting the structure of the object-oriented program's header files and class definitions correct at the expense of the real application. In this book, I give you an understanding of the solutions to the problems, not the detailed header files. I want you to understand the principles involved rather than get caught up in the implementation details. As a result, this book isn't language-specific or programming paradigm-specific; instead, it explains the underlying principles involved in the solutions that you will apply using your prior experience and expertise.

Finally, you don't have to read the whole book from front cover to back. Instead, use the table of contents and index to locate the information you need when you need it.

Conventions Used in This Book

Here are the conventions I use throughout this book:

- I capitalize the names of patterns. In some chapters, the name of the pattern is the same as the name of a key component of the architecture. In general, the pattern name is capitalized, and the name of the component is not capitalized.
- I abbreviate the names of many of the patterns discussed in Parts III and IV because they're quite long. Model-View-Controller, for example, becomes MVC. On the first use in a chapter, the whole name is spelled out, and the abbreviation is used thereafter.
- When I introduce a new term, I put it in *italics* and define it shortly thereafter (often in parentheses).
- I put web addresses in `monofont` so they stand out from the surrounding text. ***Note:*** When this book was printed, some web addresses may have needed to break across two lines of text. If that happened, rest assured that we haven't added extra characters (such as hyphens) to indicate the break. So, when using one of these web addresses, just type in exactly what you see in this book, pretending as though the line break doesn't exist.

What You're Not to Read

I've sprinkled a few sidebars around in the text. They show up as gray boxes. You can safely skip them. They contain information that I think you may find useful but that isn't required to understand the patterns or software architecture. You also can skip anything marked with a Technical Stuff icon (see "Icons Used in This Book," later in this Introduction, for more information).

Foolish Assumptions

I make some assumptions about who would read and benefit from this book. I don't expect that you're an expert in software architecture; in fact, I assume that you're pretty new to it. I do assume that you know something about writing software, however, and that you've already written some software. In particular, I assume that you've written software in some sort of team setting on a project bigger than a school project. From this experience, you'll have learned about designing with modules and components.

Because more software is changed, evolved, and maintained than written from scratch, I assume that you've experienced some software maintenance. Maintenance of someone else's (or even your own) code will have given you an understanding of the importance of modularity and good structure.

I don't assume that you're an expert in object-oriented design or any other particular design methods. The architectures in this book can be adapted to any paradigm you work in and are familiar with. Some familiarity with at least the basic terminology of objects, classes, and methods is assumed.

How This Book Is Organized

This book has five parts. Parts I and II introduce software architecture and software patterns. The next two parts present real live patterns that you can use in your software. Finally, Part V shows you where to turn next to explore the exciting world of software patterns.

Part I: Introducing Software Architecture and Patterns

To build a foundation for the rest of the book and to explain the basic concepts, Part I focuses on software architecture: what it is, how to create it, and how to document it. Architecture builds on the needs of the customer or client, so Part I also talks about the requirements that shape your architecture.

Architecture needs to be explained to those who will build the application. Even if you're the sole builder, an explanation will help you remember later what you did today. Part I introduces various ways of documenting your architecture, including simple Class-Responsibility-Collaboration cards, the basics of the Unified Modeling Language, and an outline of an architecture description document.

Part I ends with a chapter that describes the basics of software patterns. This chapter provides a foundation for the discussions in Part II of making the most of software patterns.

Part II: Putting Patterns to Work

You need to find patterns that address the problems you need to solve. Part II describes how patterns are organized and catalogued. It also presents a process you can use to find the patterns that can help you.

As you start using patterns, you'll find that you use the same patterns over and over. Part II has instructions for collecting the patterns you use most often in a private quick-reference catalog.

Part III: Creating Your Application Architecture

Part III contains eight architectural patterns that you can use in several kinds of software, ranging from distributed systems to user interfaces. Each chapter discusses a single topic. The patterns cover several different architectural problem spaces like structuring the solution, disturbed systems, and interactive systems. Chapters 13 and 14 show two different ways to solve similar problems related to user interfaces. The chapters in this part all contain implementation sections to give you an outline of the steps needed to implement the pattern's solution.

Part IV: Designing with Other POSA Patterns

Architectural patterns solve the really big problem of how to structure the entire software system. In Part IV, the focus is on smaller patterns that address smaller programming problems. These design patterns address specific elements of the software, not the whole structure. The design patterns are organized by the kind of problems that they help with, and each kind of problem is presented in its own chapter.

Solving lower-level problems that you encounter only in a single programming language is the work of an *idiom* pattern. Part IV contains an example idiom pattern that's useful in the C++ language (although the general principle is useful elsewhere as well). The patterns in Part IV also outline the steps needed to implement the solution. You'll build on your own experience to adapt these steps to your program.

Part V: The Part of Tens

Every *For Dummies* book has a Part of Tens. This book has three chapters in Part V, each containing ten tips to help you continue your study of patterns. Chapter 23 contains ten individual patterns that you should know. Chapter 24 lists ten places to look for specific patterns, including books and websites. Finally, Chapter 25 lists ten ways that you can get involved with the software patterns community, ranging from using patterns in your own development to telling people about patterns to writing your own.

Icons Used in This Book

I've used several icons throughout this book:

The Remember icon is a friendly notice of information that you should keep in mind as you're reading the text.

The Technical Stuff icon marks text that digs deeper into a concept. You can skip this material if you want.

When something is especially helpful for using a pattern or idea, I mark it with a Tip icon.

I've thrown in a few warnings, which are things that you need to be consciously aware of; otherwise, you could run into problems.

Throughout the book I provide examples of how you can use patterns in the real world. I mark that material with the Real-World Example icon.

Where to Go from Here

The book is structured so that you can jump in anywhere. If you aren't familiar with software architecture, I suggest that you start with the first part. If you already know about software architecture but aren't sure what patterns are, start with Chapter 4 and progress through Part II.

Each chapter in Parts III and IV discusses a different pattern. None of these chapters depends on your having read any other chapter. You'll see some cross-references between chapters, but they're provided to help you dig deeper and understand your options; they aren't there to point out that you should have read something else beforehand.

Part I

Introducing Software Architecture and Patterns

In this part . . .

The first part of this book introduces the underlying concepts to get you ready to use the patterns described later. I begin by giving you some background on software architecture and then discuss the basics of software patterns.

Chapter 1

Software Architecture Basics

In This Chapter

- Understanding the basics of software architecture
- Finding the problem
- Identifying requirements
- Considering your software development style

The term *software architecture* means different things to different people. To the developer, it means the structure of the system being built. To the framework developer, it's the shape of the system that is created with the framework. To the tester, it's the shape of what needs to be tested. For all concerned, it's the high-level structure of the solution to a problem that the customer or client wants solved.

In this chapter, I explain the basics of software architecture — what it is and how you get started. Knowing the problem that you're solving and the important requirements of the system are also very important, and I help you get going with these tasks in this chapter. In Chapter 4, I explain how software patterns fit into the picture.

Understanding Software Architecture

Every system has an *architecture* — some high-level structure that underlies the whole system. *Software architecture* is how the pieces fit together to build the solution to some business or technical need that your customer or client wants solved. The architecture has a purpose.

The decisions made during the creation of the architecture are truly fundamental to the system because they set the stage for all the other decisions that will come later.

Some systems' architectures are best described as a Big Ball of Mud (see Chapter 2). These systems are hard to build and hard to maintain, and they may not meet the customer's needs. Tackling the development of a software system with good software architecture will lead to a more successful result.

To an unsophisticated customer or client, *software architecture* is a meaningless term, so don't get hung up trying to explain how wonderful your architecture is. The customer wants the finished product that solves the problem at hand, not a description of the software that you'll build to solve it. (For more information on explaining software architecture to others, see Chapter 3.)

Components of software architecture

The software architecture provides the high-level view of the system you're building and must cover the following aspects:

- **Goals and philosophy of the system:** The architecture explains the goals and describes the purpose of the system, as well as who uses it and what problem it solves.
- **Architectural assumptions and dependencies:** The architecture explains the assumption made about the environment and about the system itself. The architecture also explains any dependencies on other systems or on the builders of the system.
- **Architecturally significant requirements:** The architecture points to the most significant requirements that shaped it.
- **Packaging instructions for subsystems and components:** The architecture explains how the parts of the system are deployed on computing platforms and how the parts must be combined for proper functioning. The subsystems and components are the building blocks of the architecture.
- **Critical subsystems and layers:** The architecture explains the different views and parts of the system and how they relate. It also explains the most critical subsystems in detail.
- **References to architecturally significant design elements:** The architecture describes the most prominent and significant parts of the design.
- **Critical system interfaces:** The architecture describes the interfaces of the system, with special attention to the interfaces that are critical to meet the system's requirements.
- **Key scenarios that describe critical behavior of the system:** The architecture explains the most important scenarios that illustrate and explain how the system will be used.

Architecture document

All the components in the preceding section go into an architecture document, which contains the information needed to interpret the architecture. The document includes assumptions, key decisions that shaped the architecture, how the parts of the architecture work together, and how the system will be packaged. I tell you more about the architecture document in Chapter 3.

Architecture models (views)

The software architecture has several audiences, including fellow architects, programmers, configuration managers, testers, and customers. All are interested in different information, and all look for different things within the architecture. To make your architecture useful to all these audiences, divide the architectural description into four different models or views:

- **Logical:** Maps the system onto classes and components. The logical view is directly related to the functional requirements, which I discuss later in this chapter. The logical view focuses on the parts of the system that provide the functionality and that the users of the system will see when they interact with it.
- **Process:** Explains how the parts of the architecture work together and how the parts stay synchronized. It also explains how the system is mapped onto the units of computing, like processes and threads. *Processes* are groups of tasks that together make something that can execute and perform the desired functions. The process view brings in some nonfunctional requirements (see "Defining nonfunctional requirements," later in this chapter), which aren't directly related to visible functions.
- **Physical:** Explains how the software that implements the system is mapped onto the computing platforms. The various components of the system, networks, processes, tasks, and objects are mapped onto the tangible parts of the system in the physical view. This view contains information related to the system's nonfunctional requirements (discussed later in this chapter), such as availability, performance, and scalability.
- **Development:** Explains how the software will be managed during development. The software will be written in small pieces that individuals or small teams can work on together. The development view highlights these pieces and shows how they are intertwined and interdependent. The development view reflects any limitations on the organization of the software based on limitations in the programming language, development environment, or development organization.

I tell you about diagram styles to use for each of these models in Chapter 3.

These four models of the system are usually supplemented by one additional view that describes common scenarios, tying the other views together by showing how elements within all of them work together. (Use cases, discussed later in this chapter, describe the scenarios.) This additional view is frequently called the *4 + 1 model.* Figure 1-1 shows how the parts relate. A good architecture balances all these views so that no view contains much more detail than any other.

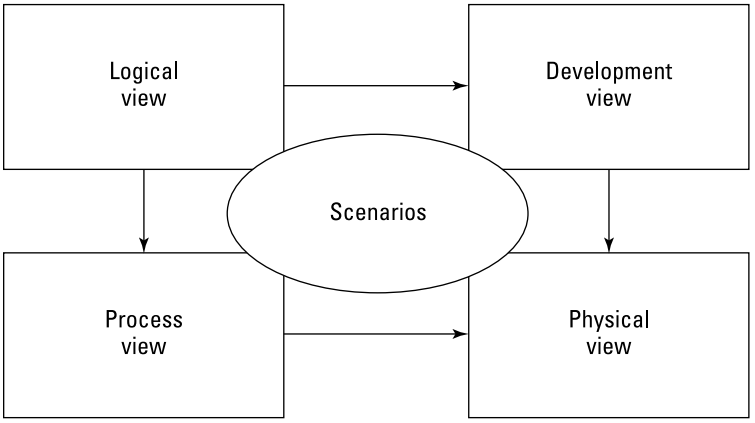


Figure 1-1: The 4 + 1 model of an architecture.

Software development methods and processes

Software development can be done in many ways. These different ways are called *methods* or *processes.* Here are a few examples:

- **Waterfall method:** In the *waterfall method,* the different phases of system development activities follow each other sequentially. The artifacts produced during development are considered to be flowing downstream and going over a waterfall between the analysis, requirements-gathering, development, and testing phases of development. Artifacts always move forward, or downstream, without repeating a phase more than once.
- **Unified Process:** The *Unified Process* is a popular process in which the various activities — such as requirements generation, development, and testing — overlap. Instead of being associated with particular work products and the tasks that create them, the phases in the Unified Process follow the life of the product, from inception to elaboration to construction and finally to transition. Within each of these phases, the activities are iterated, always focusing on the most critical aspects.

- **Agile methods:** Agile development methods are very popular today. Agile methods are an outgrowth of the Agile Manifesto (`www.agilemanifesto.org`), which declares (among other things) that there's more value in working software than in the documentation created by the waterfall method and Unified Process.

  Within the category of agile methods are a variety of methods, such as XP, Scrum, and Lean. Agile methods are also iterative, but even more than in the Unified Process, a little bit of each activity is done during each iteration.

All these methods are useful, and everything I tell you in this book about developing software architecture applies to any process you use. The only differences involve when the architecture descriptions are handed off to people working on the other parts of the process.

Identifying the Problem to Be Solved

As you define your software architecture, the most important question you need to ask is: "What problem am I solving?" A major reason why software systems don't succeed is that they don't meet the needs of the customer or client who requested the software. In other words, they didn't solve the customer's or client's problem.

In this section, I show you how to identify the problem so that you can develop a solution that both solves the problem and meets your customer's or client's needs.

Breaking the problem into the four attributes

The problems that you solve with software architectures have four main attributes:

- **Function:** Describes the problem to be solved
- **Form:** Describes the shape of the solution and how it fits into the environment of other systems and technologies
- **Economy:** Describes how much it costs to build, operate, and maintain the solution
- **Time:** Describes how the problem is expected to change in the future

Understanding these four attributes is critical to identifying the problem to be solved.

Ask the customer what he wants in a system and why he wants it. As he explains, take notes, and map them to these problem attributes.

Ultimately, the system described by your architecture must do what the customer wants, at a cost the customer is willing to pay, and on a schedule that satisfies the customer's needs.

Developing a problem statement

A problem statement is needed to understand what to build.

To show you how to develop a problem statement, I start by walking you through the process of creating an example payroll system. Follow these steps:

1. **Establish the goals of the problem-definition process.**

   Decide how long you can spend developing the problem statement and how much detail the problem statement needs to have.

   For a payroll system, you want to identify the constraints on the solution (issues that will affect its form, economy, and time) and be sure that you understand the high-level function: to get employees paid.

2. **Gather facts.**

   In the fact-gathering steps, you work with the customer or client to understand her needs, how she's satisfying that need now, and what computing platform she expects to be used in the solution. You also identify the people and other systems, known as *actors,* that will interact with the system. Your objective is to find out as much as you can about the problem, the need, and the expectations on the system.

   For the example payroll system, you would gather facts about the number of employees, how frequently they get paid, how their pay is calculated, and what potential deductions are taken from their pay.

3. **Uncover the concepts that are essential to the solution and that will shape your architecture.**

   In this step, you look for the underlying concepts in play. You uncover assumptions, equations, regulations, process models, usage constraints, and other fundamental concepts.

   For the payroll system, you discover the equations used to compute an employee's pay and determine how irregularities from normal payment are communicated with the system.

4. **Determine what the customer or client must have to be satisfied with the solution.**

   This step involves understanding the needs and expectations of the customer or client based on the underlying concepts that you found in Step 3.

What is the minimum that the customer must have to be happy with the solution you design?

The example payroll system needs to take in each employee's hours worked, to know the base rate of pay and related deductions, and to compute payment amounts. The system also needs to print checks or in some other way make payments to the employees.

5. **Write the problem statement.**

   Based on your understanding of the problem from completing the preceding four steps, you can write a problem statement that brings in the four attributes of function, form, economy, and time (see the preceding section) in a way that explains it to the customer or client.

   For the example payroll system, the problem statement is "Compute and pay employees for work done [Function] using an interactive system for entering hours worked and for making payment through direct deposit [Form]. The solution should be available in three months [Time] for the price negotiated [Economy]."

Defining the important use cases

When you have a clear idea of what the problem is, you want to refine that definition and really zoom in on what you need to do to solve the problem. An effective way to do this is to write use cases. A *use case* describes what a person should expect to accomplish when he or she uses the system. *Actors* — the people or other systems that interact with the system being designed — are the main ingredients in use cases, and I discuss them separately later in this section.

The scenarios shown in use cases connect the different views of the architecture, showing how the parts of the architecture work together to solve the problems that you've identified by describing example usage scenarios.

Choosing the functionality to capture

You write a use case to explain how some of the system's functions work and how the system interacts with actors. Use cases can be used to explain external functionality or what goes on inside the system. The external functionality is what you want to understand at this stage of your architecture development, so concentrate on the interactions of external actors with the system. As you develop your architecture, using the method I explain in Chapter 2, the internal functions of the system become clear.

Use cases can capture large functionality, such as computing weekly payroll for all employees, or small functionality, such as validating the hours worked by a single employee. Regardless of size, however, all use cases have discrete goals — specific outcomes that they describe.

To see how use cases work, consider the simple payroll system from the previous section that computes payments due and directs those payments. Figure 1-2 shows a use-case diagram for this system and the text describing the use case. Both parts are important. This use case has one actor — the employee — who is interacting with the system to update the hours that he worked.

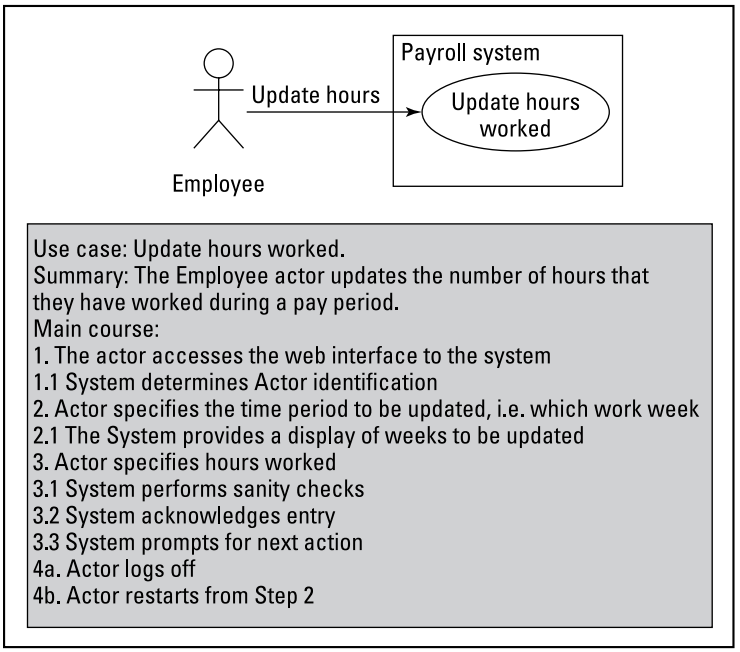


Figure 1-2: An example use-case diagram.

Use-case diagrams like the one shown in Figure 1-2 are useful for providing an overview of how the actors interact with the system and with one another.

Don't try to capture all the details in a single use case. If you do, the use case will become unwieldy.

Develop the use cases a little at a time. Start by writing a high-level use case and then add more use cases that go into greater detail.

Identifying the actors

Use cases revolve around actors. Who are these actors? Here are a few definitions:

- **Actors perform the functions described in the use case.**
- **Actors play various roles: customer, user, employee, manager, payroll clerk, and so on.**

- **Actors can be involved in many use cases.** Particular actors, like the payroll clerk, can perform different functions in different use cases.
- **Actors don't need to be human; they can be other systems.**

  When an actor is a system, use a different symbol in the use-case diagram from the one you use for humans (see the next section).
- **Nonhuman actors shouldn't be internal components of the system.** Actors are people or things that interact with the system from its exterior. For the purposes of use cases, the system is a black box, and you shouldn't include its internal functioning.

Diagramming the system

Systems have multiple use cases, so a special use-case diagram provides a high-level view of how all the actors interact with the system and serves as a table of contents for the individual use cases.

Figure 1-3 shows the use-case diagram for an entire payroll system. The payroll system is in the center, surrounded by the actors. The bubbles represent the named use cases.

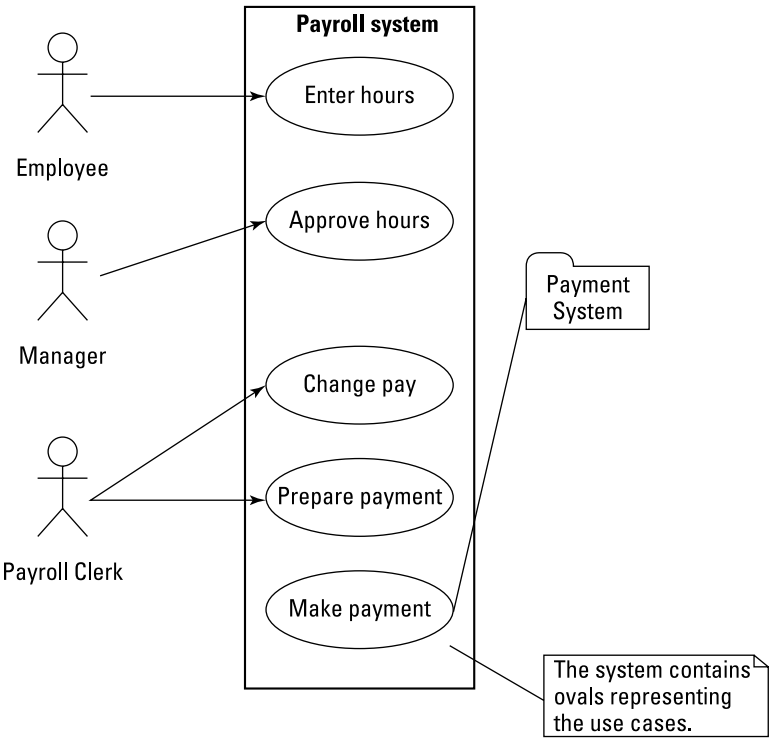


Figure 1-3: A use-case diagram for an entire architecture.

Documenting the use cases

When you begin defining your use cases, start at the overview level by identifying the most important use cases; then turn your attention to refining these use cases. Document them by using the process that follows.

These nine steps, which describe the tasks needed to develop a use case, are from *UML 2 For Dummies,* by Michael Jesse Chonoles and James A. Schardt (Wiley):

1. **Decide which use case you're going to document, and give it a name.**
2. **Sketch a diagram that shows how your actors will interact with the system.**

   For an example, refer to Figure 1-2, earlier in this chapter.
3. **Write a short summary of the use case.**

   Usually, a sentence or two is enough.
4. **Write the story of the use case.**

   The story usually begins with "The actor does *something.*"
5. **Describe the main sequence of events that will happen after the actor begins the use case.**
6. **Write down anything that must be done before the use case starts or that must be done after it ends.**
7. **Identify the other scenarios, such as error cases or alternatives.**
8. **Write the sequences of events for the alternative scenarios identified in Step 7.**
9. **Add any rules that the use case must enforce.**

   You may want to add a rule that the use case is required to validate the data input by an actor, for example.

Identifying the Requirements

When you thoroughly understand the problem to be solved, as discussed in the preceding section, you need to translate it into detailed *requirements* (the list of things that you need to include in the solution). Sometimes, you need to be formal and write down the requirements, even numbering them and tracking them through to the code. At other times, you don't need to be so formal, but you should still document the requirements. The level of detail needed in the requirements is related to the complexity of the problem and the solution; complex problems and solutions call for detailed requirements.

Architectures are created to implement and meet requirements.

You identify requirements in much the same way that you define the problem statement (refer to "Developing a problem statement," earlier in this chapter). You need to talk to the customer or client and find out what he really wants you to design and build.

Defining functional requirements

Some requirements are obvious from the customer's needs. Perhaps the customer wants the main user interface to be through a web browser, for example. Or perhaps the system needs to compute a table of values following the customer's formula, such as "compute the amount to be paid to an employee using hours worked and per-employee deductions as inputs."

Requirements like these, which define something that the system must do, are *functional requirements.* Functional requirements are represented and illustrated in use cases. When an actor interacts with the system, that interaction is made to achieve some purpose — and that purpose is the requirement.

The functional requirements show up most often in the logical view of the system (refer to "Architecture models [views]," earlier in this chapter), which shows the behavior of the individual classes.

Defining nonfunctional requirements

A system has other requirements that you won't be able to demonstrate by clicking a widget and seeing what happens. These requirements, called *nonfunctional requirements,* include things like the performance of the system, how much memory it uses, and how fast it can start.

Many lists of types of nonfunctional requirements are available, but here's the list that I like to use:

- **Changeability:** The changeability requirements all relate to how well the system can be adapted over time. The changeability-requirement family contains several subcategories:
  - Maintainability: How easy it is to maintain the system.
  - Extensibility: How easy it is to extend the system and add new functionality to it.
  - Restructuring: How easy it is to restructure the system to take advantage of new technology.
  - Portability: How easy it is to move the system to a new computing environment.

Don't allow the requirements to change too frequently — that can be a recipe for disaster. Frequent changes mean that no one will know for sure what the system is supposed to do, and they signal that the client or customer isn't sure what he or she really wants.

- **Interoperability:** The interoperability requirements describe how well the system must work with other parts of the customer's or client's computing environment.
- **Performance:** The performance requirements cover things like how fast the system must be or limits on the resources it may use.
- **Dependability:** These requirements specify how long the system must work, how secure it is, and how accurate it is. Dependability includes a large number of subcategories:
    - Reliability: How accurate the results must be and how long the system must work before it has an error.
    - Availability: The percentage of the time the system must be available for service. Availability includes fault tolerance, which specifies whether the system must tolerate faults and continue operation.
    - Maintainability: What must be designed into the system to allow it to be cared for and maintained.
    - Security: What security requirements exist for the system, what security mechanisms must be in place, what the expectations for confidentiality are, and the integrity of the system and its data.
    - Safety: Whether the public will be at risk of bodily harm from this software.

Where safety is concerned, you must look for established best practices for architecture, design, and coding within the type of system you're building. I don't talk about them in this book; you need to seek the appropriate resources.

- **Testability:** The testability requirements state what the system must do to ensure that all the requirements, both functional and nonfunctional, can be tested.
- **Reusability:** The reusability requirements specify what you need to do when designing and building the system to ensure that it can be used again. A different kind of reusability requirements specify that a certain amount of reuse be achieved within the design of the system or even that certain already-built components be used in the system.

Get SMART with your requirements

With all requirements, but especially the nonfunctional requirements, you should make the requirements SMART. This acronym reminds you that the requirements must be

- **Specific:** They describe a specific characteristic of the system.
- **Measurable:** They are testable and observable in some way.
- **Achievable:** They are realistic and can actually be achieved.
- **Relevant:** They relate to the problem that the system is supposed to solve.
- **Trackable:** They produce specific things within the architecture that you'll be able to point to later.

Here are some ways that you can identify the nonfunctional requirements:

- Find out as much as you can about the problem and how others have solved the problem.
- Extract common requirements from your reading.
- Watch how the customer uses the system that he already has, or watch him step through the process that the system will be part of.
- Listen carefully to your customer as she explains what the system will do and why it's needed.
- Ask questions!
- If you've built similar systems in the past, draw on that experience, and include the requirements that you've seen before.
- Review the problem statement with the customer so that he has the opportunity to tell you which things are important and which things are unnecessary. This review also helps refine your understanding of the problem.

You should understand the requirements that have the biggest influence on the solution architecture first, because requirements will change. Looking at the big requirements first helps get their changes out of the way early.

The nonfunctional requirements make their appearance in both the physical and process views of the system (refer to "Architecture models [views]," earlier in this chapter). The process view addresses how the execution is distributed around the system, which may be a requirement in itself or which may be related to the performance or dependability requirements of the system. The physical view of the system also shows the nonfunctional requirements

because much of a system's dependability is tied to redundancy (how the processing is distributed to reduce the effects of single points of failure). The physical view captures the performance and scalability nonfunctional requirements through information about which processing elements can be replicated to grow the system.

Reviewing the requirements

When you're developing requirements, a variety of pitfalls can make the requirements unusable or unhelpful. The requirements may describe the problem that you want to solve or require the architecture that you want to build, rather than what the customer or client really wants and needs. Also, your requirements can omit things that the customer or client thinks is essential to the solution. Review your requirements with the customer or client to avoid these pitfalls.

Here are some things you can do to make your requirements more useful:

- **Try to identify and describe the implied or hidden requirements.** In the payroll-system example, an overtime multiplier needs to be used when the hours that any employee worked in a week exceed a given threshold.
- **Validate all your assumptions.** Perhaps you assumed that no one would receive a paycheck or a related transaction for a negative payment. This assumption may not be true, however, if wages are garnished or if many fixed deductions occur.

  Never add your own assumptions without validating them with the customer. Also make sure that you identify your assumptions — and remember that they aren't facts about the system.

- **Don't overextend assumptions.** To continue the payroll-system example, you may have assumed that everyone working more than 40 hours per week would earn base pay times some multiplier. You shouldn't keep extending this assumption by assuming, say, that the multiplier is 2. One assumption is bad enough; don't make assumptions about your assumptions.
- **Avoid indecisive specifications.** Make sure that you know whether a tax rate, for example, is *x* percent or *y* percent and that you know when it applies.
- **Avoid inconsistent or conflicting requirements.** You may have a requirement to print paychecks and another requirement to provide data for a direct deposit. Which requirement is the real requirement?

- **Fit the solution to the problem.** As you find out more about what the customer or client wants, you may see that the scope of the problem statement keeps expanding, to the point that the range of possible solutions is much larger than what the customer originally asked for (see Figure 1-4). The requirements help you see what's important and what you should really build within this range of solutions.

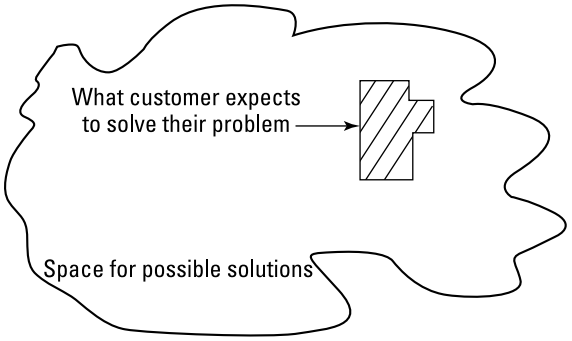


Figure 1-4: What the customer asked for may be smaller than the solution space.

Requirement do's and don'ts

Here are some good ways to ensure that your system fails:

- Don't write any requirements.
- Don't understand the usage scenarios.
- Don't understand what your customer or user really wants.
- Don't define the acceptance criteria.

By contrast, here are a few things to do (to make your requirements good and useful):

- **Describe what the system is supposed to do, and why.** Provide enough information to allow intelligent tradeoffs to be made.
- **Refrain from defining how something is to be implemented.** That definition comes into play when you are creating the architecture and design.
- **Specify technology choices only when the technology is an important aspect of the customer's problem statement.**
- **Make sure that you have all the requirements you need.** Major gaps in requirements can be critical, causing a project and/or system architecture to fail.

Choosing a Software System Style

In Chapter 2, you get down to the business of creating the actual software architecture. Before you do that, in the final step before diving in and designing the system architecture, you need to start thinking about what kind of style and shape the system should have. In this section, I highlight two aspects of system style: architectural and programming.

Architectural styles

Architectural styles define the general shape of the system. In residential housing, Cape Cod and ranch are examples of architectural styles. In software architecture, styles include Model-View-Controller and Pipes and Filters. I introduce software architecture styles in Part III.

In the different models of the architecture (such as the 4 + 1 model shown in Figure 1-1, earlier in this chapter), the views are related but also independent. You may find that you want to use a different architectural style within each view.

Programming style

You must also consider your programming style — object-based style, procedural style, or functional style, for example. Not every problem fits into every style of programming, so being familiar with multiple styles is essential to understanding the style of program you should use and choosing the right one for the problem and solution.

I won't try to explain the differences or influence your decision. Ample resources about programming in any of these styles are available, including many *For Dummies* books, and I'm sure that you have your own favorite styles. Even though this book is about patterns, however, it isn't exclusively about object-oriented programming. Patterns aren't always for objects. As you see in later chapters (specifically, Chapters 8, 23, and 24), patterns are available for a wide range of problems, not all of which relate to objects.

Chapter 2

Where Do Architectures Come From?

In This Chapter

- Getting familiar with architectural styles
- Building your own software architecture

The two major things that you'll be defining and using in your software architecture are components and services:

- **Components** are the building blocks of the system — the parts of software or the providers of functionality that you combine into your architecture.
- **Services** are the things that your components provide to the actors and to one another — the visible functionality of the system. As you divide the functionality of the system into components, you're also defining what services each component provides. The services can be internal or external to the system.

In this chapter, I tell you how to use components and services to create an architecture.

Understanding Architectural Styles

As I mention in Chapter 1, architectural style is like the style of a house. It may define a ranch house, which sprawls horizontally in one story, or a Cape Cod house, which is a two-story structure with a distinctive arrangement of doors and windows on its facade.

In software development, *architectural style* refers to the general shape of the system. Choosing the appropriate style is important because all the later design decisions are made in the context of this style and in concert with the

style. A system may have a streaming style, like Pipes and Filters (see Chapter 10), or it may have an interactive style that's shaped by Model-View-Controller (MVC; see Chapter 13). The choice of appropriate style is important to your system's success.

Elements of styles

The style defines features and rules that shape the architecture, such as the following:

- **The basic building blocks of an architecture style:** What the key elements are and how the components and services are typically named.
- **The connections between the basic building blocks:** How components communicate with other components.
- **The rules that specify how the services may be combined and used together.**
- **The family of solutions:** All streaming solutions, for example, will resemble Pipes and Filters (see Chapter 10), even if they're quite complex and diverse.
- **The contexts and problem situations in which the style is most useful.**

Patterns and architectural styles

The architectural patterns that you see in Part III of this book describe a variety of styles.

Table 2-1 lists some basic architectural styles and the patterns in Part III that help you design them. There may be more than one pattern for each style. Both MVC and Presentation-Abstraction-Control, for example, are in the Interactive Systems style. The individual pattern text contains information to help you decide which of these styles to use.

Table 2-1 Architectural Styles

Architectural Style	*Patterns*
From Mud to Structure	Layers (Chapter 9), Pipes and Filters (Chapter 10), Blackboard (Chapter 11)
Distributed Systems	Broker (Chapter 12)
Interactive Systems	Model-View-Controller (Chapter 13), Presentation-Abstraction-Control (Chapter 14)
Adaptable Systems	Microkernel (Chapter 15), Reflection (Chapter 16)

Creating Software Architecture

In this section, I take you through the creation of your software architecture. The heart of the section is a process you can use to do the actual design. First, however, I discuss some basics that cut across your architecture and the process that defines it:

- **Timing:** In the following section, I discuss when you should create your software architecture.
- **Problem categories:** The various domains of computing that influence your solution. Chapter 8 provides more information about using these categories to identify patterns that can help you solve your problem, but the topic is worth a mention here, because designing an architecture cuts across domains.
- **Layers and abstractions:** Abstraction is a very important part of computing that plays an important role in software architecture. In this section, I tell you a little bit about abstraction and the very effective technique of stacking abstractions into layers.

Finally, I give you a process you can use to shape and refine your architecture. This process is iterative: You start at a high level and work your way lower, refining the architecture as you go deeper. Some of this deep dive into the architecture layers functionality into the design. Some of the refinement comes from bringing in components and services from different problem areas.

Deciding when to create an architecture

Historically, architectures are created in the design phase or early in the development of a system. If you're using a waterfall development process, for example, creating the architecture is one of the very first things you do: You define the problem and then solve it with an architecture. If, instead, you're using an iterative software development process, such as Unified Process or agile, the architecture is typically evolved and elaborated in the early iterations in parallel with some low-level design and coding. As iterations of the architectural development become stable and complete, the other steps, such as design and coding, can begin. Each iteration may include more refinement to the architecture in conjunction with further design and coding.

Many methods of agile software development call for having potentially shippable products at the end of each iteration. In the early iterations, some of the shippable products are products that you'll use internally, such as the architecture description, tools, and frameworks.

Defining the architecture is a very important step that shouldn't be skipped. Skipping it can result in sets of components being used in incompatible ways, which means that you have to back up and redo work.

Skipping the architectural-development stage can result in a Big Ball of Mud (a.k.a., Shantytown or Spaghetti Code). This well-known software pattern (`www.laputan.org/mud`), by Brian Foote and Joseph Yoder, describes the alternative to a system that has a well thought-out architecture that guided development.

Identifying problem categories

As you see in Chapter 8, patterns are available for many problem categories, which are subject areas or domains. When you're developing your architecture, you need to solve problems from many domains.

In a payroll system, for example, you need some help from the database domain to record employee payment history. You also need help from the interactive system domain to input current working hours so that you know how much to pay the employees.

The solutions to real problems cut across different domains of computing.

In Chapter 8, I tell you about a way to identify the patterns that will apply in solving a problem partially by zooming in on different problem categories.

Defining layers and abstractions

Chapter 1 discusses the 4 + 1 model of software architecture: the logical, process, physical, and development views, plus scenarios or use cases. When you create a software architecture, however, you aren't really going to build five things or even four things. The objective is to build one system with a single software architecture. Those different views are just different ways of looking at the same architecture. Each of these views shows an abstraction of the architecture that focuses on a particular element of the design.

Layers

Sometimes, your system or its environment has explicit layers of functionality. If you're building a communications system, for example, you need to be familiar with the Open Systems Interconnection (OSI) seven-layer communications model, shown in Figure 2-1, because your system will be most successful if it fits into these layers. In the OSI model, the layers show up in the logical

view of your architecture. I tell you more about this kind of layering in Chapter 9, the first chapter of this book's collection of architectural patterns.

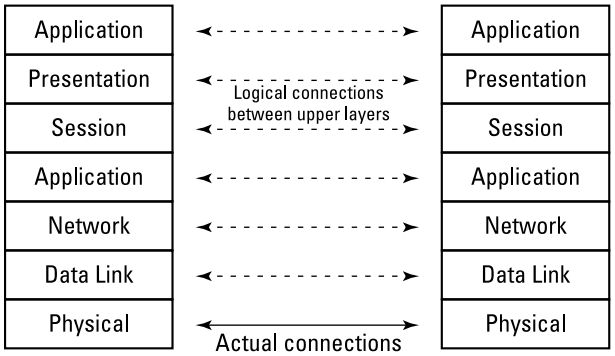


Figure 2-1: The OSI model.

In other cases, the layering may involve layers governed by the physical system — typically referred to as *tiers,* not layers. An example in the web world is the three-tier model, where the presentation server, the application server, and the database servers are implemented as different physical devices. These three tiers communicate by passing information to the adjacent tiers. Three-tier architectures frequently are distributed as shown in Figure 2-2 because the components in the different tiers have different processing needs.

Because layering is such a fundamental architectural principle, it is also covered in Chapter 9.

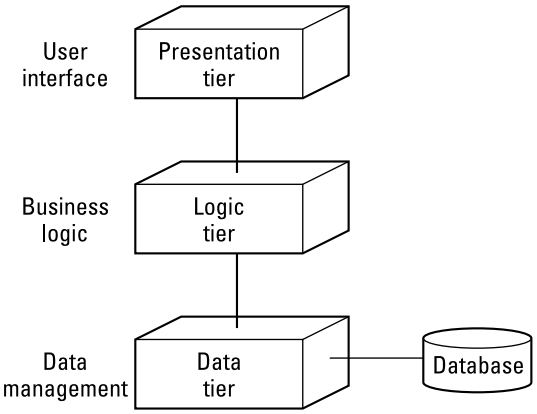


Figure 2-2: A three-tier model based on physical separation of functionality.

Abstractions

What these different ways of layering the system have in common is that they're abstractions. An *abstraction* is a way of describing something in general terms that leaves out the details of any specific implementation. In the examples in the preceding section, the abstraction is a general layer that doesn't describe how the functionality is implemented or even precisely what it does; those details are abstracted away.

In the next section, you see that abstraction is one of the important techniques that help you build better architecture. Abstractions are important to software architecture because they allow you to talk about what the system does in general terms before you've worked out all the low-level details.

In the same way, the three-tier architecture separates the functionality by abstractly grouping the presentation server and keeping its functionality separate from the functionality of the database servers. This is done because they have different processing or storage needs, and the boundary of an abstraction can be drawn around the layers.

Being able to abstract the essence of a functionality and project it onto the solution are important skills that you must have as a software architect.

Employing enabling techniques

There are a number of fundamental principles for constructing software that I call *enabling techniques*. These techniques are independent of the particular methods that you use to create software, such as waterfall, agile, or Unified Process, all of which I discuss in Chapter 1.

Balancing the trade-offs involved in using the enabling techniques helps you create an architecture that balances the functional and nonfunctional requirements of the system, which I also discuss in Chapter 1.

The enabling techniques are

- **Abstraction:** Abstraction is the ability to extract the common, general parts from a particular entity. You use abstraction to define a common component that will be adapted to several specific situations in your system. This technique is exactly what I discuss in the preceding section.
- **Encapsulation:** Encapsulation is grouping related elements to preserve the boundaries of the abstraction. You use encapsulation to keep related functionality together instead of mixing unrelated functionalities.

- **Information hiding:** Information hiding keeps information that clients don't need to know hidden from them so that it's protected and the clients don't misuse it. Encapsulation is frequently used to implement information hiding.
- **Modularization:** Modularization handles system complexity by breaking the system into parts with well-defined boundaries. This technique is especially useful as you design software architectures because they frequently are too big to be implemented efficiently as single entities. Modules can contain one or more components, as I talk about elsewhere.
- **Separation of concerns:** Within the system, unrelated responsibilities should be separated. You use separation of concerns to define elements that perform specific functions rather than elements that perform a variety of functions.
- **Coupling and cohesion:** Coupling is how different modules in the system relate to one another. Cohesion is a measure of how related the objects and functions within a module are to each other. High cohesion and low coupling lead to systems that are easy to modularize and build.
- **Sufficiency and completeness:** Every component should be sufficient to include all the characteristics that are needed for useful and efficient interaction with other components. Every component should also capture all the important characteristics making it complete.
- **Separation of policy and implementation:** Keeping the implementation of algorithms free of system-context-related information simplifies reuse. You use this technique to design parts of the system to deal with context-related, or policy, information and other parts to implement abstract algorithms.
- **Separation of interface and implementation:** This technique separates the interface that clients use from the implementation of the functionality that the clients expect. It makes reusing the implementation easier and promotes information hiding (discussed earlier in this list).
- **Single point of reference:** Avoid inconsistency by defining the items within the software architecture only once. Achieving a single point of reference depends on your programming environment, because some languages, like C++, make achieving this principle difficult. Although C++ also requires a single point of definition, it needs declarations to appear in several places.
- **Divide and conquer:** Divide a problem or solution into smaller parts that are easier to solve or implement. You use this technique often as you work with large problems.

Patterns for enabling techniques

Many of the patterns in Part III and IV help with applying the techniques described in this chapter. I tell you about patterns starting in Chapter 4. You can wait to see the actual patterns until later, but if you want to read ahead, the following table points you to patterns later in this book that help you achieve the benefits of the specific enabling techniques. Of course, these aren't the only patterns that address these enabling techniques.

Enabling Technique	*Pattern*
Abstraction	Layers (Chapter 9)
Encapsulation	Forwarder-Receiver (Chapter 21)
Information hiding	Reflection (Chapter 16), Whole-Part (Chapter 17)
Modularization	Layers (Chapter 9), Pipes and Filters (Chapter 10), Whole-Part (Chapter 17)
Separation of concerns	Model-View-Controller (Chapter 13)
Coupling and cohesion	Client-Dispatcher-Server (Chapter 21), Publisher-Subscriber (Chapter 21)
Sufficiency and completeness	All the patterns (Chapters 9–22)
Separation of policy and implementation	Strategy*
Separation of interface and implementation	Bridge*
Single point of reference	No specific pattern
Divide and conquer	Microkernel (Chapter 15), Whole-Part (Chapter 17)

** This pattern is available in Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional)*

Keep these enabling techniques in mind as you define your software architecture (see the next section). They'll help guide you as you divide your problem into smaller pieces that you can solve and then combine back into your overall software architecture.

Designing your architecture

Developers design their architectures in many ways. In this section, I present a process that I find useful.

To design your own software architecture, follow these steps:

1. **Select a component to be refined.**

   When you're just getting started, the first component that you select is the whole system. You zoom in to details in future iterations.

   For the component that you're refining, you first define the goals of the component. Use the requirements and the problem statement (both described in Chapter 1) as input to understand what the component is supposed to do.

2. **Identify the requirements of that component and the requirements for its interactions.**

   What other parts of the system or external world does it interact with? Use cases (refer to Chapter 1) help you understand the interconnections and the service it needs from other parts of the system.

   Lay out the high-level information flow between these components. Think about what parts of the component are responsible for different parts of the architecture. Think about the different processing steps that are required and what component of the system will perform them. If you're designing with objects, in this step you will brainstorm the "classes" of the system.

   What you identify in this step is the general shape of the architecture.

   Class-Responsibility-Collaboration (CRC) cards are very handy tools to use during architectural refinement to record the components, what they do, and what other components they work with. I discuss another use of CRC cards in Chapter 3.

   a. **For each class or component, write a CRC card.** The card indicates the name of the item at the top, as shown in Figure 2-3. Below the name, on the left side, list the responsibilities of this class or component, such as "Remember hours worked for the period." On the right side, write the names of the other objects or components that this one interacts with, such as "Payment calculator" or "Hour register."

Name: Hour Store	
Responsibilities: • Remember hours worked during pay period	Collaborators: • Hours register • Payment calculator

Figure 2-3: CRC card.

b. **Use the scenarios in your use cases to guide you through the CRC cards.** Step from card to card. Each card invokes the responsibilities listed on the next card. Pay attention to what's missing. Check the scenarios that are documented in your use cases to see whether you've defined all the classes or components you need to execute the scenario. If you find others, write cards for them.

3. **Search for an existing architectural style or pattern that fits the requirements and interactions that you identified in Step 2.**

   In Chapter 8, I give you a set of steps to use to find particular patterns. Also, check the patterns in Part III of this book to see whether any of the patterns listed there match the structure and interactions that you've identified.

   If you can't find a category that matches your problem perfectly, look in a category that's similar.

4. **Use the pattern that you've matched to your problem to guide the arrangement of your classes and components.**

   All the patterns contain an explanation of a proven structure for the interactions between classes or components. The patterns also may contain the kinds of dynamic interaction (messages or calls) that need to be exchanged.

   One reason to use patterns as inputs to your software architecture design process is that patterns describe the trade-offs in the solution. They tell you more than just how to do something; they also tell you about the other options and why their trade-offs aren't as good.

   In this step, you use the extra information of the proven software solution you found in Step 3 to shape the classes and components you found in Step 2.

5. **Iterate through the components, repeating Steps 2–4 for each one.**

As you pick the next component to design, you may be greatly tempted to pick the one you're most interested in. That reaction is only natural. You should pick the next-most-important component instead, however. Make your selection based on critical functionality needed by other components or based on the hardest component to design.

It may take more than one iteration of this process to come up with an architecture that truly satisfies all your requirements. Creating software architecture isn't always easy, but the more architectures you design, the easier the process gets and the better the results are.

Documenting your work

Take some notes while you're defining the architecture. Keep a record of the key decisions. Sketch out how the parts fit together. These notes will help you down the line as you document your architecture (see Chapter 3).

If you make any assumptions while you're developing your architecture, keep track of them so that you can validate them with your customer or client. When you make decisions after trading off one alternative with another, make notes to help you remember why you selected one alternative over another.

If you reject some ideas because they won't work for this problem, write down the ones you rejected and the reasons why you rejected them. You'll be surprised how often someone will second-guess the decision. If you have the reasons why you rejected a different approach at your fingertips, you can save everyone a lot of time.

Chapter 3

What Do Software Architectures Look Like?

In This Chapter

- Exploring basic UML diagrams
- Using tools to draw your architecture
- Describing your architecture to other people

It's not enough just to create a software architecture; you also have to be able to explain it to other people. Diagrams help you convey the shape of the system and present the different viewpoints that different people may be interested in.

In Chapter 1, I introduce the 4 + 1 model: logical, process, physical, and development views, plus scenarios or use cases. In this chapter, I tell you how to use the Unified Modeling Language (UML) to diagram these views.

Examining UML Architectural Models

You can visualize and explain an architecture in many ways. The most common approach in use today is the Unified Modeling Language, or UML for short. In this section, I provide an overview of how UML is used to illustrate software architectures.

Choosing a diagram style

UML has many specific symbols and conventions. I won't tell you about all of them, because you can find books related to the topic, such as *UML 2 For Dummies,* by Michael Jesse Chonoles and James A. Schardt (Wiley).

Here are a few of the diagram styles that I find most useful for describing architectures:

- Class
- Interaction
- Deployment
- Packaging
- Use case

In the next section, I tell you when I use each of these diagram styles. Later in this chapter, in the section "Explaining Your Software in an Architecture Document," I provide detailed information about using diagrams in your documentation.

Showing different views

You create diagrams to explain the software architecture to your team members, colleagues, and management and to help you remember what you've designed with the types of UML diagrams just mentioned. These diagrams fit into the 4 + 1 model that I describe in Chapter 1 and that you see in Figure 3-1, which shows the types of UML diagrams that are best suited to the different views.

The following sections discuss the correlations between the views and the diagram types.

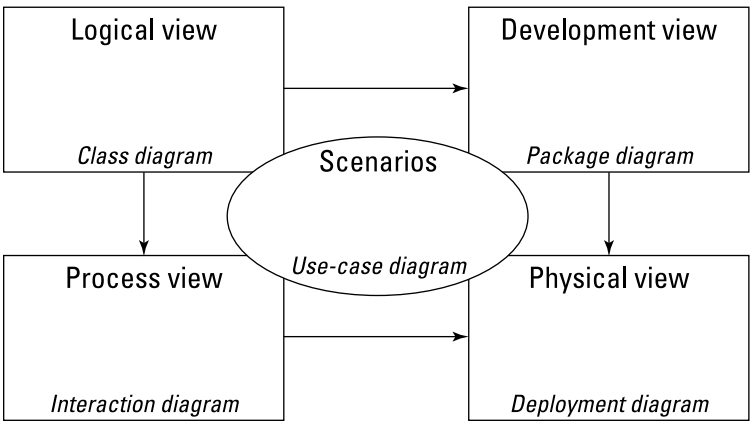


Figure 3-1: The 4 + 1 model with UML styles.

Logical view

When you want to show how individual classes and objects are related and connected, you use a class diagram. Class diagrams show how the individual classes and objects fit together *statically* — when they aren't actually executing. The diagrams show the logical relationships — usage, composition, inheritance, and association — between the classes.

Most people involved with system development use the logical view, because it explains what the system is supposed to do, and how. The logical view is useful for explaining to your customer or client that the functionality he or she desires is reflected in the architecture. It describes the functional requirements that I describe in Chapter 1. You can point to parts of the class diagram that show the logical view to explain where the system implements specific functionality.

Process view

When you want to show how components of the system exchange information during execution, you use an interaction diagram. This diagram shows the dynamics of the system — that is, how messages flow between the tasks and processes. The process view shows many nonfunctional requirements as well.

Testers and integrators use this view because it explains how the parts of the system exchange information and react.

Physical view

To show where the packages fit on the various physical parts of the system, such as networked computers, you use a deployment diagram. You may have multiple deployment diagrams showing different configurations, such as the development and test configuration and the production configurations.

The people who are building the networks of computers that will become the system use the physical view. Anyone who's working on communication between the parts of the system or the parts of the system and the external world also will be interested in the physical view.

Development view

When you want to show how parts of software are related and dependent on one another in the development view, create a packaging diagram. The development view shows module and subsystem boundaries. The packages show the groupings of classes or other components that will be developed and distributed together.

Programmers and managers are interested in the development view. Also, anyone who is involved with creating the development environment will be looking for the development view.

Scenarios and use cases

When you want to explain what your system does or how someone would interact with and use it, you create a use case, as described in Chapter 1. A use-case diagram shows how the various actors and use cases relate to your architecture in different usage scenarios.

The scenarios and use cases are the most important things you'll share with your customer or client. The scenarios explain how the system is going to behave and how it performs the required actions. The scenarios also explain the user interface, describing the ways that the users interact with the system and how the system interacts with other external systems.

Scenarios and use cases are the glue that binds the other four views together. The scenarios explain how the user interacts with the classes, components, processes, and subsystems that are shown in the other views.

Working with UML Diagrams

The preceding section tells you which UML diagram is most appropriate to capture each view in your 4 + 1 architectural model. In this section, I show you the basics of these diagram styles.

The goal is to build a system that solves the customer's or client's problem — not to produce pretty UML diagrams.

Creating class diagrams

Class diagrams show you the static relationships between parts of your system in the logical view. They also may show a conceptual model of your system. I call these parts *classes* to match the name of the type of diagram, but they can be any parts or components of your system — they don't need to be the classes of an object-oriented design.

I use class diagrams to model the conceptual view of the system because they're most appropriate for an architectural view, but you also can use class diagrams to show the specification of the system or the implementation. You use these other views in later phases of system development.

When drawing a class diagram of your architecture, concentrate on the most important things. Although you may want to draw the diagram all the way down to the implementation in the parts you know best, you need to provide an overview of the key areas, instead of focusing on one particular area.

Sometimes you need to go into additional detail about what happens inside a class. For this task, you use state charts or state transition diagrams, which I don't cover in this book. To understand and describe the general shape of the system's architecture, you don't need to go to this level of detail. You can find more information in *UML 2 For Dummies,* by Michael Jesse Chonoles and James A. Schardt (Wiley).

Associations

The static relationships include how the different classes are associated. Class diagrams are static because they show the relationships that exist between parts of the system even when the system isn't executing. Drawings of the system's dynamics show how messages or information flow between the static components of the system. These are discussed in the "Showing the interactions" section, later in this chapter. These associations can be any of several types:

- One class may inherit from another class or refine another class.
- Classes interact and communicate with one another.
- Classes work together to provide functionality more complex than any class can provide by itself.

Associations are shown as lines connecting the classes (see Figure 3-2). Each association has two roles — one at each end of the line. Associations can be named, which you should do if naming them makes the relationships between the classes more understandable.

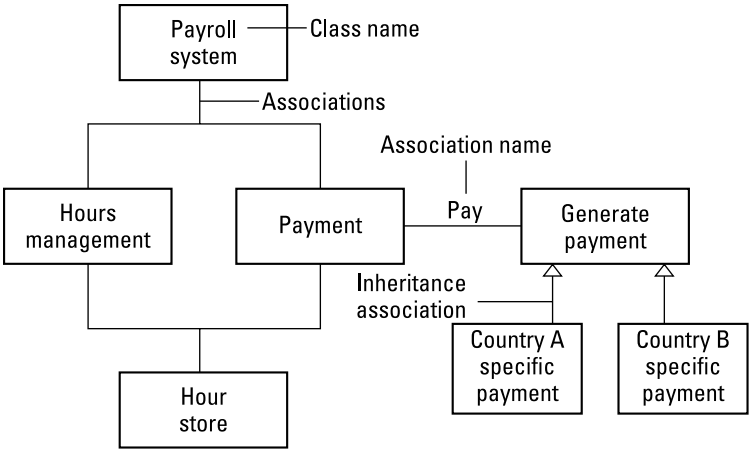


Figure 3-2: An example class diagram showing basic classes and associations.

Working with CRC cards

UML isn't your only choice for documenting the information that you put into UML class diagrams. An older method that's really simple and well suited to group or object-oriented development involves using Class-Responsibility-Collaboration (CRC) cards. These cards are a different, simpler form of class diagrams. (Here, I use *class* to mean a group of software components that may or may not make up an object-oriented class.)

To create a CRC card, draw a horizontal line near the top of an index card; then draw a vertical line from that line to the bottom of the card. Fill in the card as follows:

- At the top, write the title of the class.
- In the bottom-left corner, write the responsibilities of the class.
- In the bottom-right corner, write the names of the classes — the other cards — that this class collaborates with.

The following figure shows example CRC cards for a payroll system.

Payroll system	
• Provide overall payroll interface • Send UI to input hours or compute pay	• Hour management • Compute pay

Hour management	
• Communicate with user to input hours worked	• Hours store

Hour store	
• Store hours worked by employee	None

CRC cards, as noted in Chapter 2, are an excellent way of brainstorming architecture with your colleagues. The cards are very flexible and let you create new classes easily. The cards can be spread out and rearranged on a table, and if you find that the class on a card is no longer needed, you can throw away the card. You can put your CRC cards into the architecture document along with your UML class diagram, if you want. I discuss the architecture document in the last section of this chapter.

Class multiplicity

A class diagram also can show that one class is really several of the same class, or *class multiplicity.* A class in the architectural diagram could show the relationship of one e-commerce website to multiple customers or a personnel database to many employees, as shown in Figure 3-3.

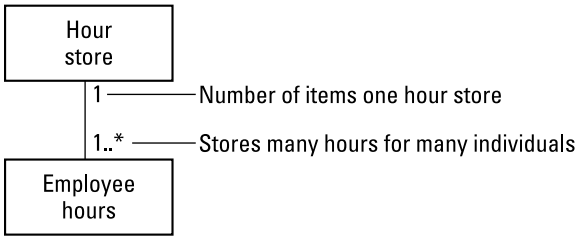


Figure 3-3: A class diagram shows class multiplicity.

Attributes and operations

Class diagrams commonly show *attributes* (things that the class is responsible for remembering or storing). At the architectural level, however, attributes aren't used often. Some examples of attributes that you may see in a class diagram at an architectural level are

- Data elements to be stored
- Roles that this class satisfies
- Important requirements that this class satisfies, especially nonfunctional requirements and references to supported use cases

Class diagrams also traditionally show *operations,* which are the kinds of things shown in the responsibilities part of a CRC card (see the nearby sidebar "Working with CRC cards"). The responsibilities may be things like

- Tasks and operations completed in conjunction with a use case
- Important functionality provided by the class

Attributes and operations are shown as separate sections in the box for a class. These sections are separated by horizontal lines, as shown in Figure 3-4.

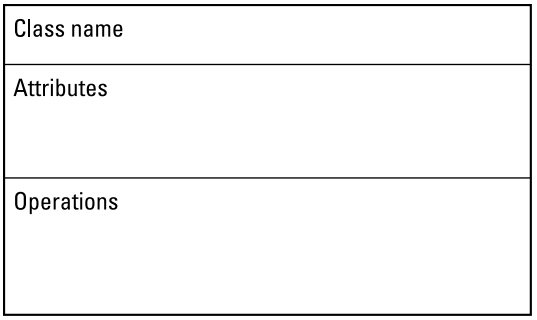


Figure 3-4: A complete class symbol.

Showing the interactions

Interaction diagrams show how parts of the system communicate with one another during execution. This type of diagram is part of the process view, which also shows how the parts of the system are synchronized. Interaction diagrams show that the parts talk — a fact that's captured in the class diagram (refer to "Creating class diagrams," earlier in this chapter) — but they also show the sequencing and ordering of those communications. Individual interaction diagrams usually show the interactions of only one use case.

Interaction diagrams were called *collaboration diagrams* in UML 1.

At their most basic, the different components are shown as boxes at the top of vertical lines. Arrows between the lines represent messages sent from one component to another, as shown in Figure 3-5. Time is represented as vertical timelines, so a message near the top is sent before a message drawn near the bottom of the diagram.

Interaction diagrams can be quite complex, however, with multiple communicating components and many messages flowing every which way.

You won't need an interaction diagram for every component pairing. Pick only the scenarios that have the highest risk or the most valuable interactions.

The arrows showing messages between components can be angled down slightly, or they can be horizontal. You can use this type of diagram to show special things such as iteration, in which a message loops back to the sender. Figure 3-6 shows some of the special things you can display in an interaction diagram.

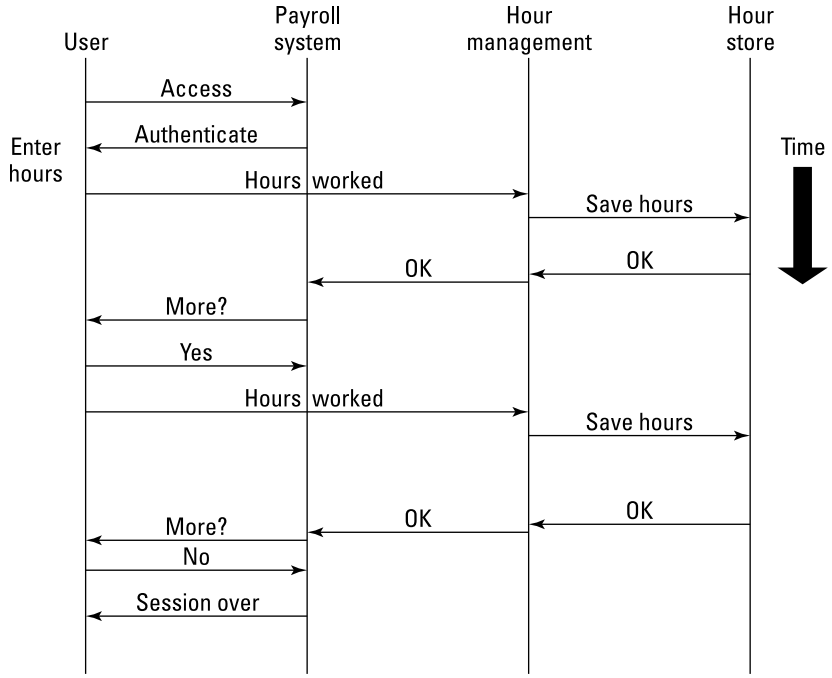


Figure 3-5: A simple interaction diagram.

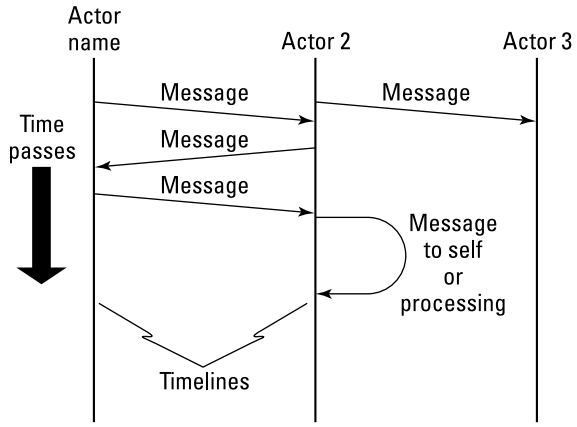


Figure 3-6: The parts of an interaction diagram.

Deploying your system

To show how you plan to get your software on the hardware that will run it, you use a deployment diagram as part of the physical view. This type of diagram shows the physical relationships between your computational devices — in other words, your pieces of hardware. The architecture that you're creating may be deployed on PCs that talk to centralized servers to perform the query functions, as shown in Figure 3-7.

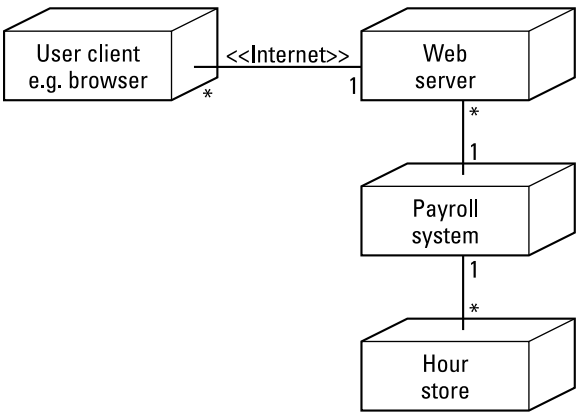


Figure 3-7: A deployment diagram.

Deployment diagrams are similar to class diagrams, which I cover earlier in this chapter, because they show connections between different physical parts of the systems that are required to achieve the overall system functionality.

The components that are shown in the deployment diagrams are the physical modules of code — the packages in the package diagram, which I tell you about in the next section. The relationships between the deployment diagram packages show that the deployed packages are connected, either via a network or internally.

You can have more than one package, or module of code, per deployment component.

To develop your deployment diagram, start at the system level, and divide the functionality into parts.

Packaging up the software

Your architecture is composed of parts that contain components that are closely related — or *highly cohesive* and that work together as a unit. They're only loosely coupled with other parts of the system. (Coupling and cohesion are discussed as enabling techniques in Chapter 2.) These components should be developed together, packaged together, and deployed as a single part. The UML packaging diagram shows how the parts relate in the development view.

The packaging diagram shows classes and the dependencies between them. It's very similar to the class diagram described earlier in this chapter but is interpreted differently. Packages are shown as rectangles with tabs on the top, like tabbed file folders. The names of the packages are shown within the tabs. Figure 3-8 shows a simple example of a packaging diagram.

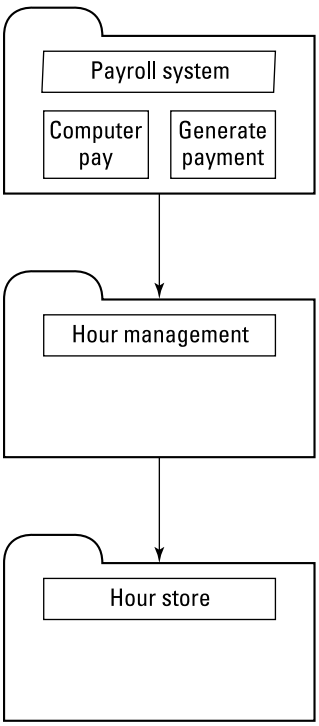


Figure 3-8: A simple packaging diagram.

Keep the following rules in mind when you create packaging diagrams:

- **Packages own their content.** Individual components or classes can't be included in more than one package.
- **Packages are dependent on other packages if there are dependencies between two components within the packages.** Packaging diagrams don't reduce the dependencies between components, but they show them so that they won't be forgotten.
- **Unlike the dependencies introduced by a compiler, the package dependencies aren't transitive (in other words, Package A doesn't need Package C unless Package B is included).** In Figure 3-9, which shows a packaging dependency package, Hour Management depends on Hour Store, but the bigger Payroll System depends only on Hour Management; it doesn't also depend directly on Hour Store.

Using use-case diagrams

Use-case diagrams, which I introduce in Chapter 1, are simple and straightforward. Figure 3-10 shows the key elements of these diagrams.

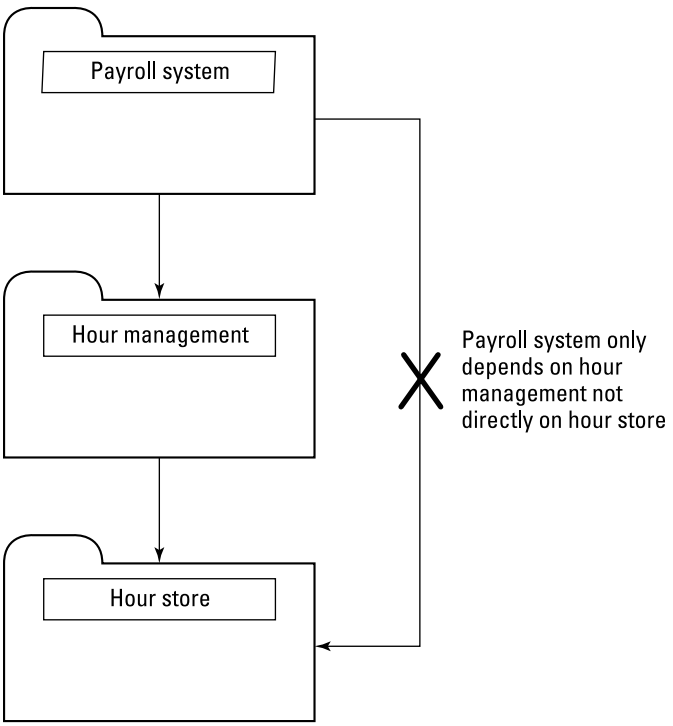


Figure 3-9: Package dependencies aren't transitive.

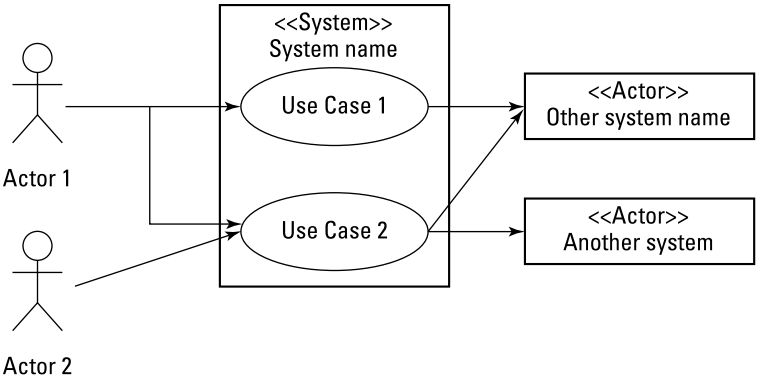


Figure 3-10: The key elements of use-case diagrams.

Use-case diagrams are important because they show how actors interact with the system and how the use cases embody the functional requirements.

You can use use-case diagrams to show how the user and other key actors, such as maintainers, interact with the system. They help you remember who the system is being built for and how the actors plan to interact with the system.

Choosing Your Design Tools

While you develop software architecture, you have many discussions with customers, other architects, and designers about the problem, the expectations for the system, and the requirements. During these discussions, you come up with some initial ideas and start taking notes for your architectural document, as I discuss in Chapter 2. Eventually you'll show your architectural designs to these people, however, so you need to be sure that they can understand your diagrams — a task that's made much easier when you use a standardized notation language like UML. After all, the old saying "A picture is worth a thousand words" applies in architecture, too.

Many software tools enable you to draw your designs. Having a good electronic repository of architectural views — such as class diagrams, interaction diagrams, and packaging diagrams — will help you keep your documentation in good shape. The tools help you connect the pieces and keep the big picture straight in your mind.

The tools dedicated to UML diagramming actually give you the capability to build your model of the whole system. The diagrams that are essential to explain your architecture are just views into the model.

Commercial software-development tools

Several commercial tools are available to create the diagrams described earlier in this chapter, like IBM's Rational Software Architect. Some free products, such as Astah (see "Free UML tools," later in this chapter), are also available in commercial versions that offer more features. You may not have access to the commercial tools because they're quite expensive, but that's okay, because you can easily get by with general drawing tools or free UML drawing tools.

Free UML tools

In addition to the commercial software development and UML tools there are noncommercial tools you can use.

One free software tool that you may find useful is Astah Community (`www.astah.net/editions/community`), the free community version of a commercial product. This tool is easy to use and comes with diagram styles for all the UML diagrams that I discuss earlier in this chapter. It enforces some of the UML rules and provides excellent capabilities for keeping your diagrams tidy. Figure 3-11 shows an example screenshot of the tool.

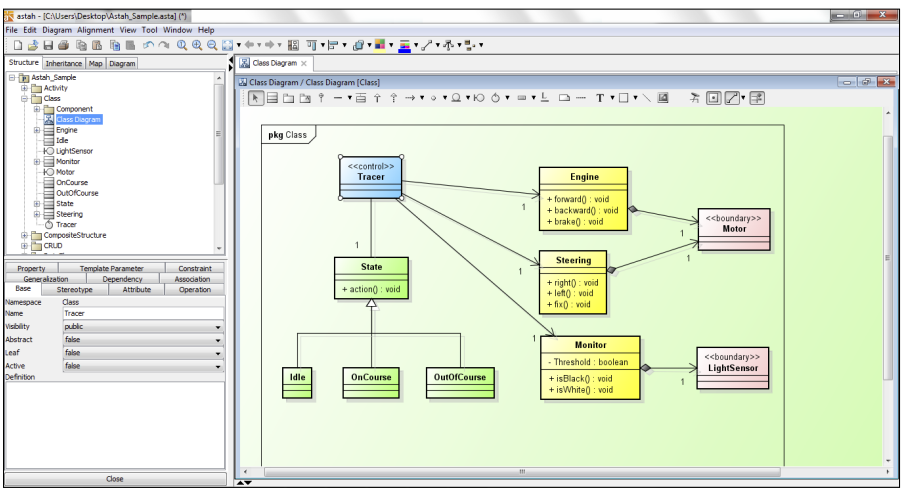


Figure 3-11: Astah Community.

Reproduced by permission of Change Vision, Inc.

Another free tool is Dia (`www.live.gnome.org/dia`), which is available under the General Public License (GPL). Dia also is easy to use. It comes with a smaller toolkit than either Visio or Astah Community, but it has all the basic building blocks that you need to create the diagrams I describe in this chapter. Dia doesn't enforce the rules as rigidly as Astah Community does — which (as I note for Visio in the next section) can be a good thing or a bad thing. Figure 3-12 shows an example screenshot.

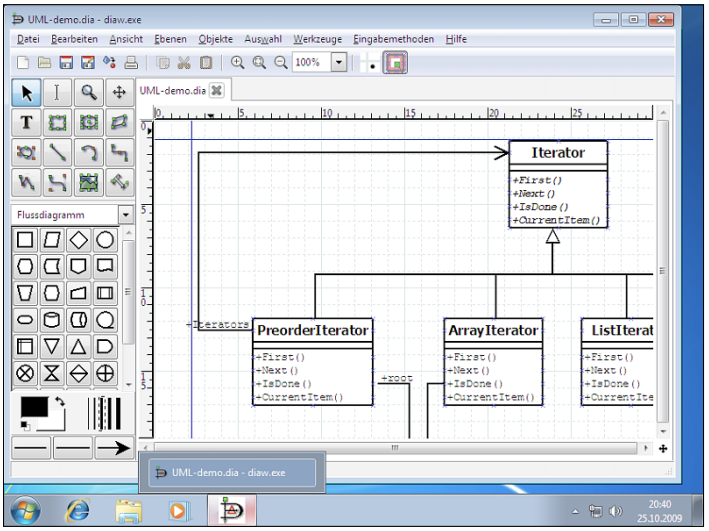


Figure 3-12: Dia.

Reproduced by permission of Steffen Macke

General drawing tools

You can also use general-purpose drawing tools to create your UML diagrams. Microsoft Visio, for example, includes a UML model drawing type and a complete set of shapes to help you create the diagrams mentioned in this chapter. Visio provides a blank canvas, so you can draw whatever you want, even if you deviate from the UML standards. The freedom to draw exactly what you want has a downside, however: some people may not understand what you're showing through your nonstandard UML.

An even more free-form approach is to use a tool like Adobe Illustrator. With this general-purpose drawing tool, you have complete freedom to draw the diagrams. Illustrator doesn't include the template libraries offered by the UML commercial tools or the free tools, so you'll really be starting your drawings from scratch.

Finally, you can use a variety of noncomputer tools, which can be anything you can draw with, down to a pencil and paper or a whiteboard and marker.

Explaining Your Software in an Architecture Document

There's more to software architecture than the pretty pictures you create to make your 4 + 1 model. You need to incorporate your UML diagrams in an architecture document that explains why the architecture is being proposed and how it meets the customer's needs. This document should explain the key abstractions that you used, as well as the patterns that helped you design the architecture. (I begin telling you about patterns in the next chapter.)

Organizing the architecture document

A very complete architecture document contains quite a few sections. The actual organization of your document may vary, but a typical table of contents looks like this:

1. Architectural Goals
2. Architectural Significant Requirements
    2.1 Functional
    2.2 Nonfunctional
3. Decisions and Justification
4. Key Abstractions/Domain Model
5. Software Partitioning
    5.1 Logical Component Model
    5.2 Process Model
    5.3 Physical Component and Layers
    5.4 Development Model
6. Deployment Model

Not all your architectures will need this much documentation. You should be guided by two factors: what your audience wants to see and what your team needs to move forward.

Filling in the sections

What goes into all these sections? The description of your architecture that I've been telling you about in these first three chapters. I give you specific contents of each section in a moment.

You don't need to complete all the sections before you get started building the system. The contents can be supplied during each iteration of your development rather than all at the beginning. By incrementally adding content, you build a living document that always accurately reflects the architecture.

Section 1

In Section 1, you describe the functionality for which you're building the system — in other words, the problem statement (refer to Chapter 1).

This section also is a good place to add a glossary so that everyone knows how the project uses terminology.

Section 2

In the next section, you talk about any requirements that are particularly significant. The requirements that tell you what functionality you need to create go in Section 2.1, and the most important nonfunctional requirements, such as overall performance or dependability, belong in Section 2.2.

Your use cases and scenarios will be split between Section 1 and Section 2.1 because the use cases describe what the system does and how it should do it.

Section 3

At the end of Chapter 2, I tell you to write down your decisions to help you remember why you made the choices that you did. If you have a hunch that someone is going to question a decision, beat him or her to the punch by using Section 3 of your architecture document to explain the rationale behind your decisions.

Section 4

Section 4 of the document is where you explain the big building blocks that you came up with and the major abstractions. You also use this section to describe the development view because it shows the big pieces and how they'll be built by the developers. If you want to remember any important trade-offs, you should put them in this section too. Any critical subsystems that are instrumental to meeting the goals and requirements should be introduced in Section 4 as well.

Section 5

You describe the 4 + 1 model of the system in Section 5 of the architecture document, which is the heart of your architecture document technically. The technical audience will be relying on the information here about the static shape of the system (Section 5.1), the dynamics and interactions of the system (Section 5.2), the processing architecture (Section 5.3), and how all these components come together in the development environment (Section 5.4). In all these sections, you should highlight the interfaces among the system you're building, its internal components, and the outside world.

Section 6

In Section 6, in addition to your deployment diagrams showing the physical view, you put instructions and notes about how to roll out the system and put it into production.

Chapter 4

Software Pattern Basics

In This Chapter

- Knowing what patterns are — and what they aren't
- Seeing what goes into a pattern
- Recognizing the major pattern styles

As I'm sure you noticed, the title of this book includes the word *pattern.* So, you probably figured that I'd get around to telling you what patterns are and aren't. I do just that in this chapter. Patterns appear in many forms, but every pattern contains a proven solution to a problem you may encounter.

What Patterns Are

A *pattern* is a recurring design element in the world or in software. An old saying applies: "Twice is a coincidence; three times is a pattern." A *software pattern* is a solution to a software design or coding problem that has been useful at least three times — a requirement known as the Rule of Three (see Chapter 5). The recurrence shows that the pattern is a common solution that works over and over again.

Patterns result when multiple people look at multiple bits of designs or code and notice similarities in the way the design or code is structured. Someone then takes the next step of taking the time to write down the pattern in a way that makes it usable for the many others who haven't had a chance to look at those initial designs.

You start to note similarities in how something is implemented and used when you've seen something happen at least three times. The same basic structure is seen in all the instances, but there are variations. The Composite pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional), for example, is used repeatedly to combine different

kinds of objects, yet the structure is still that of a Composite. This is okay. A pattern can be seen and used in hundreds of places but never be precisely the same in all the places.

You may ask, "Are the bad things that I see in the world over and over again patterns?" Well, they are, but the pattern community focuses on the constructive good things rather than the bad ones. Software patterns solve software problems. Some people talk about anti-patterns, which document the failing bad things, but I won't talk about them here. They usually tell the reader *not* to do something, which frequently isn't a helpful suggestion.

In the following sections, I focus on various attributes that define patterns. Later in this chapter, I take the opposite tack, describing what patterns are not.

Reusable designs

It's great to reuse software that you've written sometime in the past, reuse the class or function that you created for another project or another class, or use some open-source software that has a whole community behind it, fixing bugs and adding new features. If the interfaces work out, you can use the software without changes. But even if you have to adapt it with a few minor changes, you still realize the benefit of not having to re-create it from scratch.

Patterns allow reuse in the same way because they provide reusable solutions to problems. Sometimes you can reuse a pattern as it is; other times you need to make a few minor changes. In most patterns, the reuse is at the design level. Patterns usually don't contain code that you can cut and paste; instead, they contain design information that you build into your design. Even though you can't reuse code, you can reuse the design that you turn into code, which still streamlines your work.

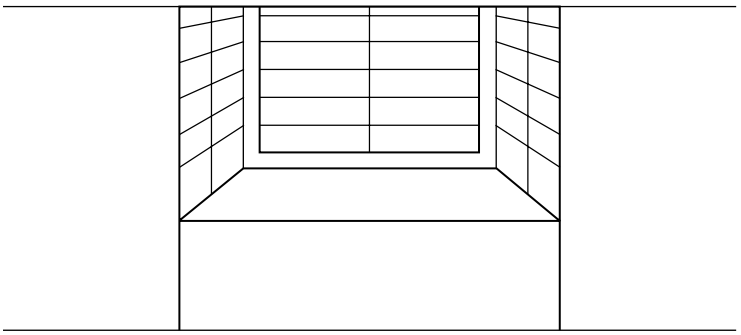
Reusable design elements are modular, flexible, and usable more than once. Open-source archives contain software that's reusable — a fact that many people take advantage of. Most open-source licenses allow you to reuse small parts of software rather than the whole, and you can even customize it to meet your specific situation.

Another attribute of reusable design is that it can be communicated. When you create a reusable design, you want others to use it, too; good communication is essential to that. You need to tell everyone else how to use it, how to configure it, and how it works. Patterns help you describe your designs.

A pattern contains enough information to help you re-create the design and understand why the solution is the best one for the situation.

Patterns do two things at the same time: They describe something and describe how to make that thing. Figure 4-1 shows two patterns that describe the appearance of a solution and give you insight into how to build that solution: Window Place, from building architecture, and Leaky Bucket Counter, from fault-tolerant design.

A software pattern is a description of a modular proven solution to a design problem with enough information that the reader can adapt it to unique situations. A key element of this definition is that the pattern contains enough information for you to read and understand the problem and solution, and see when and how you can adapt it to your own unique situation.



Window Seat

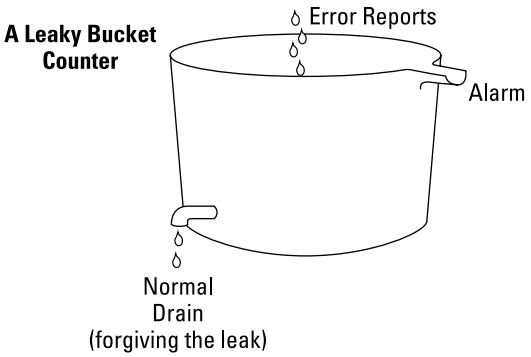


Figure 4-1: A pattern shows both a thing and the rules for making that thing.

A few favorite real-world patterns

A Pattern Language (Cambridge University Press), by architect/urban planner Christopher Alexander and his colleagues at the Center for Environmental Structure in Berkeley, California, contains 253 patterns related to the design of the real, physical world, such as buildings. The patterns range from Independent Regions to Things from Your Life and everything in between.

Following are some of my favorite patterns from this book. I encourage you to pick up a copy and browse through it. You'll find good ideas that you've observed around you, as well as good ideas you can employ in the future.

- Alcoves
- Garden Growing Wild
- Light on Two Sides of Every Room
- Pools of Light
- Site Repair
- Window Place
- Zen View

Proven solutions

Patterns describe proven solutions — those that have already stood the test of time. They help you see what has worked in the past, and they help you avoid reinventing the wheel. Why invent a new solution when you can reuse a proven solution? You should concentrate your time and effort on the new problems that haven't been solved yet. If you reuse a proven solution, you have a good chance of achieving success more quickly than if you try to invent a solution from scratch.

Just as patterns help you avoid having to solve the same problem repeatedly, patterns help you avoid making the same mistakes over and over. A pattern explains why it's the correct solution by explaining obvious, less-effective solutions and why those solutions won't work. A pattern also tells you when the solution is appropriate and when you should look for a different pattern elsewhere.

Educational tools

Each new area of programming that you start working in has some basic information that you need to acquire quickly in order to become effective. Patterns can help you explore a new computing domain and decrease your learning time. Reading the patterns of the new domain gives you a head start on understanding what issues and trade-offs are most interesting and useful.

The patterns help you see the effective solutions and understand the vocabulary (see "Architectural vocabularies," later in this chapter).

Domains as different as banking, e-commerce, telecommunications, high-performance computing, and enterprise computing have their own sets of patterns, but they all use the same common architectural patterns, which I discuss in Part III. Some of the patterns are usable in more than one of these domains. (The Risk Determination and Defense in Depth security patterns, for example, are useful in the security, reliability, and safety domains.)

System guides

Patterns provide guidance at the architecture level (see Chapter 2), showing you how to structure your system. The patterns that appear later in this book map into different architectural styles.

Architectural vocabularies

Patterns define a shared vocabulary, which is one of their most important benefits. When someone says "Singleton" or "MVC," for example, you'll know what those terms mean. A common architectural vocabulary helps everyone on the design team speak the same language.

Explaining your design to someone is easier when you both know the common building blocks of software design. When you've been drawing the design repeatedly, but the other person still doesn't understand it, try again using common principles and patterns that both you and your listener know. Patterns make the explanation easier.

Patterns also give you a richer vocabulary, so that you don't have to design based on the primitive constructs of a language or methodology such as pointers or classes. Seeing the patterns in a situation allows you to say things like, "We'll have a composite of the equipment classes that handles the way that the hardware is combined and is presented as the model that we'll use MVC to present to the user." This type of language is especially useful in structuring object-oriented (OO) code, in which the relationships of the classes and objects require careful design.

Patterns first entered the software realm through the OO community, when some of the early OO experts noticed the same structures and behaviors occurring again and again. They found the pattern format to be useful for explaining recurring structures so that other people could reuse the designs,

rather than reinvent them. Today, of course, patterns aren't exclusively about objects or OO design; they cover a wide range of topics and situations.

Repositories of expertise

Patterns are good for capturing expertise because they describe why something should be done in certain ways. Experts acquired their knowledge through years of experience, and they call upon this knowledge when they're asked to solve problems. When this knowledge is captured in a pattern, reading that pattern lets you see a problem through an expert's eyes.

To attain the benefits of expertise contained in patterns, you need to be familiar with lots of patterns — especially the ones that address the kinds of problems you usually face. There are lots of ways to do this, including reading the pattern literature and making your own handbook or catalog to note the interesting new patterns that you find. (You find out how to create your own catalog in Chapter 7.)

What Patterns Are Not

Sometimes, knowing what something *isn't* helps you see it more clearly, and that's certainly true of patterns. Here are some of the things that people frequently confuse with patterns:

- **Patterns aren't frameworks (and vice versa).** Frameworks are bits of reusable code, and patterns are textual explanations of frameworks, showing how they were built and how they can be customized.
- **Patterns aren't algorithms.** An algorithm describes a repeatable, terminating process of well-defined steps that produce some result, but it doesn't explain when those steps should be used or why they're the appropriate solution. A pattern includes this rationale. Likewise, an algorithm won't describe the trade-offs and analysis that go into deciding that the steps are the correct steps, but a pattern tells the reader why it's the right ordering of steps.
- **Patterns aren't patents.** These two terms sometimes confuse non-native English speakers because they sound similar. A patent grants the exclusive right to produce a useful product, whereas a pattern is a description of how to solve a problem in a way that has proven to be effective. Patents must be novel, but patterns describe proven practice. The goal of a patent author is to allow the patent holder to be the only one to build the patented invention. The goal of a pattern writer, on the other

hand, is to share the knowledge of how to reuse and achieve the benefits of the pattern's solution.

- **Patterns aren't exclusively for OO design.** Software patterns first gained prominence through the OO community (see "Architectural vocabularies," earlier in this chapter), but many non-OO patterns are available.
- **Patterns aren't universal problem solvers.** Patterns give you insight into the minds of experts and help you understand the things that those experts know, but they won't make you an instant expert yourself. You still need to apply your own creativity, intelligence, and taste to determine how patterns fit together and fit into the big picture.

  Also, patterns offer solutions to recurring problems, but the problems that you face won't be identical to the problems described in each pattern, so you'll have to know when a pattern is applicable to your situation and when it has to be adapted for your situation. (Chapter 8 gives you the details on choosing the best pattern for specific situations.)

If the only tool in your toolbox is a hammer, the whole world looks like a nail. For that reason, patterns shouldn't be the only tools in your toolbox. You also should know relevant algorithms, have access to appropriate frameworks, and have other resources that you can call upon to solve your software problems.

Looking Inside Patterns

Now that you know what patterns are and what they're not, you're ready to take a look at what's inside a pattern. Every pattern contains, in one form or another, the following information:

- The title of the pattern
- A statement of the problem
- The context in which the problem exists
- The forces or trade-offs involved
- The proven solution

The solution balances the forces to solve the problem in the context.

In the following section, I look at these parts of a pattern in detail, as well as some other common pattern sections. (See Chapter 8 for a discussion of the how to evaluate a pattern's usefulness.)

Title

The first thing you see when you look up a pattern is the title. A good pattern title gives you a sense of what the pattern does and how it does it, and it may even enter the architectural vocabulary (see "Architectural vocabularies," earlier in this chapter).

Sometimes, a pattern has a few alternative titles — other names for that pattern that you may already know or (for an unpublished pattern) various titles that the author is still considering.

Problem statement

Each pattern should contain a clear statement of the problem you're solving. The problem should be specific to the context, and it should be succinct. Good patterns solve small problems instead of attempting to solve big problems like world hunger. These smaller patterns, examples of which are shown in Figure 4-2, can be combined to solve the big problems. You likely won't have all the required ingredients to solve the big problems.

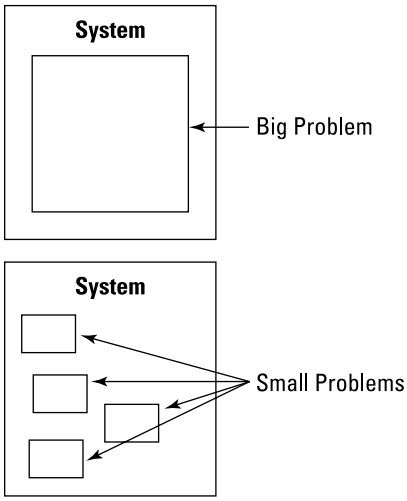


Figure 4-2: Big and small problems.

The problem statement should be relevant to the context (described in the following section), specific, and easy to understand. When reviewing patterns, you'll frequently skim the problem statements to see whether they fit the problem that you're trying to solve.

Generic problem statements like "Solve world hunger" or "Do the right thing" aren't very helpful. If you see patterns that have generic problem statements, they probably won't be detailed enough to help you solve your own problem.

The problem statement explains what the problem is and what needs to be solved.

In Part III, I explain the problems that the patterns address in the description of the example problems. In Part IV, the problem statements are in the general descriptions of the patterns.

Context

Context is one of the most important sections in a pattern because design problems don't exist in isolation; they exist in some *context.* Patterns don't exist in isolation, either; they build on the environment of the problem and upon one another. Sometimes it won't make sense to apply the solution of one pattern until another pattern has been applied. The context of a pattern describes any of these precondition patterns, as well as any other applicable preconditions.

In Part III, I explain the pattern contexts in the description of the example problem. In Part IV, the contexts are in the general descriptions of the patterns.

Sometimes when you're reading a pattern, you'll see that the context is actually presented ahead of the problem. This order shows that the context is setting the stage for the problem. At other times, the pattern author thought that the problem should appear first, so he or she put the context after the problem. Either order is okay.

The pattern's context includes the following:

- **Environment:** The context explains the environment in which the problem exists. If the context doesn't describe your environment, you may not be able to use the pattern even though the problem it solves is the same as yours.

  The context is very important for defining where a problem exists. Consider these scenarios, in which the solutions to the problems are very different if the contexts are different:

  - Ensure data consistency when updating a database: The solution will be quite different depending on whether the context is a single CPU system or a loosely coupled distributed system.

  - Maintaining a minimal memory footprint as objects in memory are created and then no longer used: This isn't a problem in Java because of the built-in garbage collection. If the context is that you're using C++, for example, a pattern like Counted Body (see Chapter 22) can solve this problem.

- **Preconditions:** Perhaps some relevant preconditions or solution constraints limit the applicability of a pattern. Here are some examples of preconditions:
  - Streaming data rather than being all available simultaneously
  - Reusable core that will be built up and extended later
  - Distributed environment
  - Heterogeneous environment
  - Java (or C++, C#, Haskell, Lisp, Ruby, . . .) language
  - Tight memory constraints or small memory footprint
- **Assumptions:** Sometimes the same problem has different solutions in the same system. The developers may have access to internal tools or the innards of a system, while the integration testers or field support engineers don't have that same access. In other words, their context of available tools is different. Assumptions about the environment and the target audience should appear in the pattern's context.
- **Constraints:** The constraints on the system that you can't change or control also appear in the pattern's context. A problem may exist only in a particular programming language.

Forces

The forces section is the heart of the pattern. This section usually isn't present in design documents and isn't specified in an algorithm, which sets patterns apart from those descriptions.

Forces got their name because pattern authors took the language of architecture and building design for the initial definitions of patterns and their parts. Figure 4-3 shows that they're called *forces* to reflect the real-life forces of wind and gravity acting on a building.

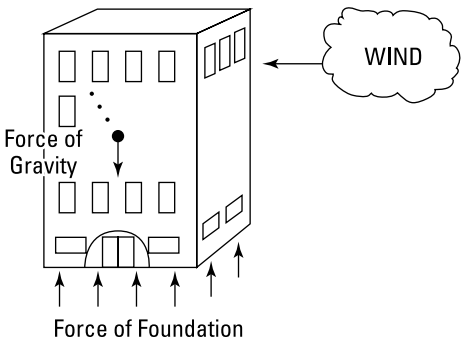


Figure 4-3: Christopher Alexander was an architect, so he used the language of architecture and engineering.

Solving programming problems requires making trade-offs. A solution may increase reliability but reduce performance, it may be faster but take more memory, or it may be easy to use or easy to build. These trade-offs must be addressed to arrive at the best solution. The forces section of a pattern explains the trade-offs and discusses how to balance them to achieve the best solution.

Of course, "best" depends on your context. Even though something is written as a pattern, with the best solution, you still need to make sure it fits your problem's context and solves your problem, which may mean adapting the pattern.

A good forces section in a pattern should lead you to understand that "this problem is hard to solve," because the section explains the pros and cons and the trade-offs, and shows why the less-good choice really is less good.

The context section contains the things that can't be changed in applying a solution. The forces section contains those things that you can change and must be balanced (traded off against each other) to arrive at a solution.

Sometimes, there is an obvious solution to the problem. But if the obvious solution isn't the best solution, the forces section explains why. The obvious solution when you have components that communicate, for example, is to hard-code the physical addresses, but this solution isn't good because it limits future flexibility.

In Part III, I describe the forces through the example problem's introduction. In Part IV, the forces are in the pattern's description.

Resolving not solving

Sometimes, the solution of a pattern is referred to as a *resolution* instead of as a solution. The reason stems from the start of patterns in the world of architecture. When physical structures are being built with patterns, they're subjected to real forces — specifically, the forces of the wind and gravity, among others. When they look at a system of bodies and the forces that apply to them, mechanical engineers talk about *resolving* the forces. What this means is balancing and equalizing all the forces to achieve stability. This is the same as when software architects and designers balance the forces that are their trade-offs to achieve a good and stable design.

In Parts III and IV, I list the pattern solutions in between the example scenarios or pattern description and the guides to implementation. In most of the patterns, there is a heading containing the word *solution*.

Solution

The solution section of a pattern explains how to solve the problem that exists in the context with the forces mentioned earlier in this chapter. The section starts by clearly stating the action you should take to resolve the problem. Then the discussion proceeds to give you enough information so that you can build something.

The solution explains how to solve the problem in the stated context. The solution statement should be specific in how it solves the problem. It shouldn't be general, offering unclear guidance like "Do the right thing."

Table 4-1 shows two examples of mismatched problems and solutions. In one example, the solution clearly matches the problem and will be helpful and useful. The other example shows that sometimes the problem and solution don't work together — the sign of a poorly written pattern.

Table 4-1 Mismatched Problems and Solutions

	Problem	*Solution*
Good	Resources for correcting errors are limited.	Correct only errors that reoccur.
Bad	Each part of the software has an unknown number of errors.	Count errors in each part of the system.

Also, the solution may not fit exactly what you want to build, but patterns support reuse at the design level. You may need to customize the solution to fit your situation.

Other common sections

Patterns also may contain other sections. Different pattern authors have found that different sections work best for their writing styles. They've also found that patterns that target different audiences may require some variation in the type of information that they provide.

Here are some of the most common "other" sections used in patterns:

- **Consequences:** The consequences of applying the solution to the problem may be spelled out in the solution section, or they may appear in a separate section. Every solution has some consequences. The good consequence is that the problem is now solved. The solution may also provide some other benefits to the system, such as making it easier to expand in the future or maybe easier to maintain. But solutions can introduce liabilities as well. Maybe the software will be harder to maintain, or extra classes will need to be created, or new problems will be introduced.

  There's generally a one-to-one correspondence between the forces and the consequences.

  In Parts III and IV, the consequences are listed in the "Exploring the Impacts" sections.

- **Sketch:** Many patterns contain a sketch or two. It's really useful if the pattern contains a sketch. The sketch may be of the solution, or it may be a sketch of the problem. The sketch may be UML diagrams or simple block diagrams of the solution. Figure 4-4 shows some typical sketches.

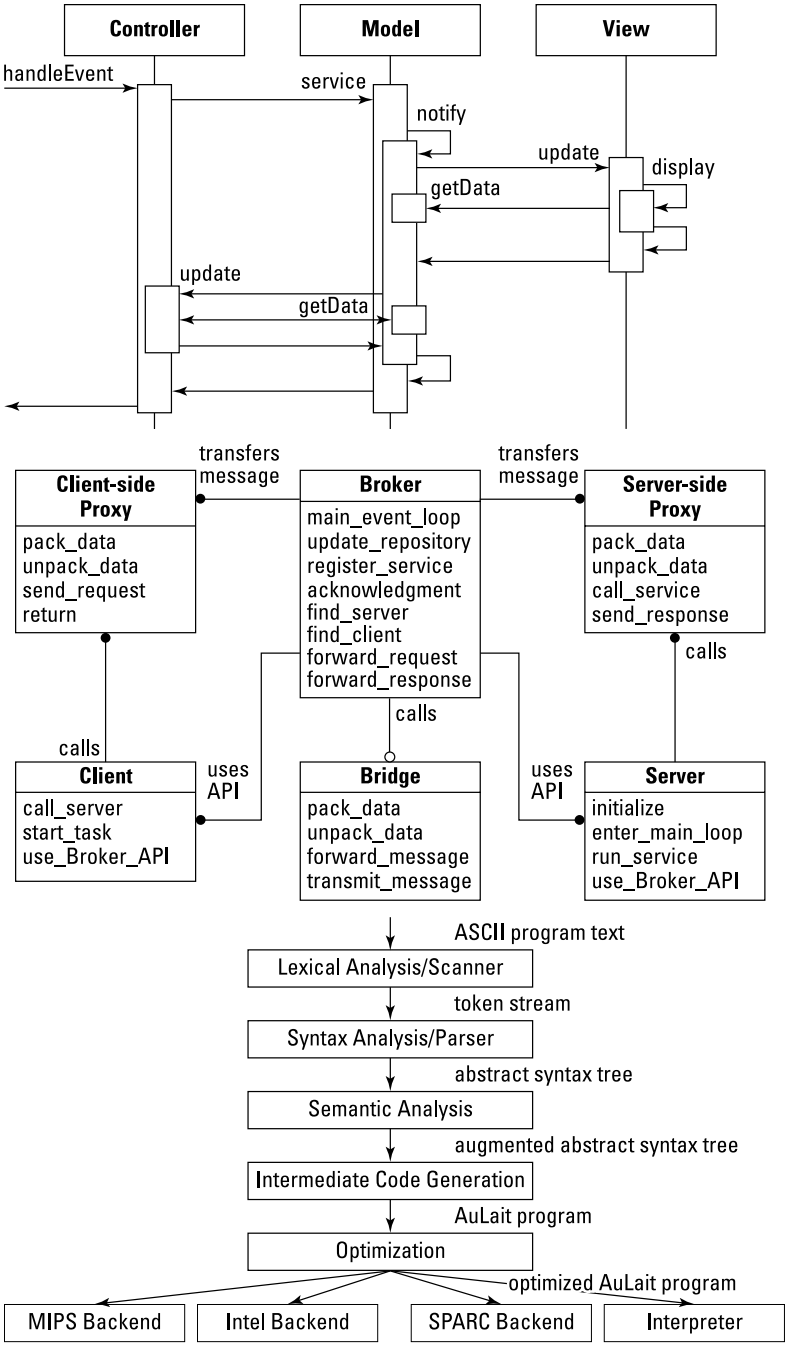


Figure 4-4: Some example sketches from *Pattern-Oriented Software Architecture: A System of Patterns.*

Reproduced with permission of John Wiley & Sons, Ltd.: Pattern-Oriented Software Architecture: A System of Patterns, 1996, Buschmann et al.

The idea of a sketch is to get you thinking visually about how the solution can be structured. The sketch also provides another view of the solution, which helps make it clearer.

The patterns in Parts III and IV of this book contain many sketches scattered throughout.

- **Resulting context:** Just as the problem existed within a context, the solution creates a new context, the *resulting context,* in which new problems exist. The resulting context also describes what the system looks like after the pattern has been applied and the problem has been solved. This section is closely related to the consequences section, and sometimes the consequences are included in the resulting context instead of in a separate section.

  The resulting context section includes a description of new problems that may have been introduced through the application of the solution, as well as pointers to the patterns to address these problems.

  In Parts III and IV, the resulting context is summarized in the "Exploring the Impacts" sections.

- **Rationale:** Some patterns include a rationale section that helps explain in plain language why the pattern's solution is the best solution for the problem.

- **Implementation:** Many patterns include an implementation section, which gives you instructions on implementing the pattern, sometimes as step-by-step instructions that you can follow.

  In Parts III and IV, the implementation details are in the "Implementing" sections.

- **Sample code:** Many patterns include some sample code. This section may show code related to any part of the pattern.

- **Known uses:** Known uses (at least three) also may be present in the pattern. Known uses help you to see that the pattern really has been used in a situation like yours.

Understanding the Patterns Used in This Book

Patterns come in many styles. The pattern styles that many people know are derived from *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley), and *Design Patterns: Elements of Reusable Object-Oriented Software,* so I look at those styles in this section. For a discussion of other pattern styles, see Chapter 5.

The Design Patterns pattern style

Design Patterns: Elements of Reusable Object-Oriented Software contains 23 patterns that improve the quality of OO design. The patterns are widely applicable, but the examples they contain use only the C++ or Smalltalk programming language. These patterns are widely taught in college courses; as a result, they're the first and only patterns that many people ever see.

Many books translate the *Design Patterns* patterns into different languages or different contexts, such as *Design Patterns For Dummies,* by Steve Holzner, PhD (Wiley); *The Design Patterns Java Workbook,* by Steven John Metsker (Addison-Wesley); and *The Design Patterns Smalltalk Companion,* by Bobby Woolf (Addison-Wesley).

The sections used within *Design Patterns: Elements of Reusable Object-Oriented Software* are shown in Table 4-2.

Table 4-2 *Design Patterns* Pattern Sections

Section	*Used For*
Pattern name and classification	The title of the pattern and the category to which it belongs
Intent	A brief explanation of what the pattern does
Also known as	A list of other names that you may associate with this pattern
Motivation	A scenario to illustrate the problem that the pattern solves
Applicability	An explanation of when the pattern can be applied
Structure	A graphical representation of the relationships of the classes that solve the problem
Participants	An explanation of all the components seen in the structure
Collaborations	How the participants work together to solve the problem
Consequences	The good and bad effects that this solution has on the problem and design
Implementation	Tips, techniques, and steps for implementing the pattern
Sample code	Actual code snippets to help you implement the pattern in C++ or Smalltalk
Known uses	Real systems that are known to implement this pattern
Related patterns	Closely related patterns that may complement and enhance the solution or be alternative ways to solve the problem

The Pattern-Oriented Software Architecture pattern style

The *Design Patterns* style isn't the most popular one used by today's pattern authors. Today, the style introduced in *Pattern-Oriented Software Architecture: A System of Patterns,* is used more widely. That book, where the patterns in the book you're reading were first published, uses the pattern style shown in Table 4-3. The sections differ slightly from the *Design Patterns* style, generally by being more explicit about where certain kinds of information can be found.

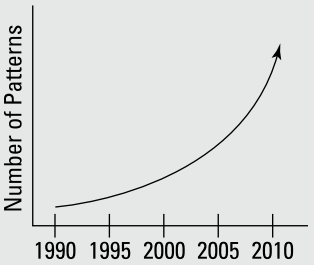
Table 4-3 *Pattern-Oriented Software Architecture* Pattern Sections

Section	*Used For*
Name	The title of the pattern and a short summary
Also known as	A list of other names that you may associate with this pattern
Example	A real-world example that shows that the problem is real; a running example through the rest of the pattern sections
Context	Where the pattern applies
Problem	The problem that the pattern solves
Solution	The fundamental solution of the pattern
Structure	A detailed explanation of the structural aspects of the pattern
Dynamics	Typical runtime scenarios
Implementation	Guidelines for implementing the pattern
Example resolved	How the original problem was solved and key points that were not raised in the other pattern sections
Variants	Similar related variants or specializations of the pattern
Known uses	Real systems that are known to implement this pattern
Consequences	The good and bad effects that this solution has on the problem and design
See also	Patterns that are closely related or that solve similar problems

Patterns past, present, and future

Patterns started gaining recognition and widespread acceptance in the mid-1990s with the publication of *Design Patterns: Elements of Reusable Object-Oriented Software,* followed by *Pattern-Oriented Software Architecture: A System of Patterns.* People started writing other patterns and publishing them in articles, books, and conference proceedings. In the '90s, patterns were primarily for doing OO design, but they evolved to cover everything from aspects to windowing and everything in between, helping all kinds of software get built in all kinds of ways.

In 2000, Linda Rising catalogued more than 1,000 widely available published patterns. The following figure illustrates the growth curve in just a decade. I'm sure that her 1,000 is only a fraction of what Rising would find today if she repeated her efforts.



Patterns will continue to be written because practitioners will always need tips, tools, and proven solutions. Also, acceptance of patterns in the academic world will continue to increase. Already, degrees have been conferred based on study of patterns, and many professors have received tenure for publications including patterns. These trends will continue because patterns fill a niche that is not otherwise filled.

Chapter 5

Seeing How Patterns Are Made and Used

In This Chapter

- Finding out how patterns are created
- Writing patterns of your own
- Understanding how patterns document designs

A pattern is more than just a good idea that someone had in the shower one morning; it's a proven solution. It's also a way to explain how system architecture is designed.

Likewise, developing a pattern for other people to use is about more than just writing it down. In this chapter, I show you what goes into creating a pattern, from idea to expert review. I also explain how you can use the patterns that you've written to document your design.

Creating Patterns

Writing your very own patterns isn't as hard as you might think, but it's not trivial either. You need a good idea — which is some solution that you've seen somewhere — and you need to dig deep to identify the trade-offs and forces that make it a hard problem (see Chapter 4) in order to understand why this is the best solution to the problem. After you get your pattern written, you should have it reviewed and share it with your colleagues so they can learn from what you've done.

For an in-depth look at all the sections that a pattern can contain, see Chapter 4.

Coming up with the idea

Anyone can be a pattern author, including you. In fact, many patterns are written by people just like you — people who see a solution more than once and think that someone else can learn from a written explanation of that solution.

Consider the pattern Model-View-Controller (MVC), which I discuss in Chapter 13. Most people know this pattern, which is used countless times each day in building and using systems. Writing a pattern like this one requires skill in the domain and skill to say to yourself, "I've seen this before. What problem does this pattern solve, and how can I explain the trade-offs that make it the best solution?"

Sometimes, a pattern isn't created from thin air but mined from software in the same way that gold is mined from the earth. The software pattern already exists. It just needs to be extracted from the surrounding system and combined with the instructions for building it (see Figure 5-1), as I describe in "Writing the pattern document," later in this chapter.

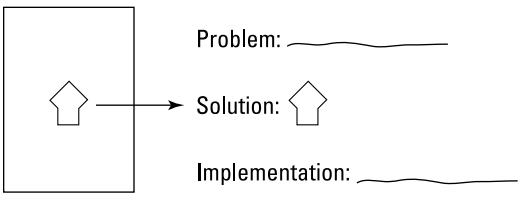


Figure 5-1: Extracting and explaining a pattern.

The pattern community wants to give authors credit for writing patterns so that they feel good about their contributions and write more patterns. Often, a pattern is associated with the name of the person who took the time to write it ("Pattern by Frank Buschmann, 2011," for example). That person may not be the actual author or inventor of the original construct that was implemented and proved to be a useful solution — just the person who wrote the pattern document.

If you know who invented a key software concept that figures into a pattern you're writing, you should record that person's name and keep it within the pattern description.

Most software, however, is anonymous. Only a few people know the names of the developers who wrote important software such as Skype or Microsoft Windows. Patterns acknowledge those people by mentioning inside the patterns the known uses and where you can see the pattern at work.

Confirming the Rule of Three

The pattern community believes in the *Rule of Three,* which means that something must be used at least three times successfully before it can be called a pattern. These uses are described in the "Known Uses" section of the pattern document (see Chapter 4).

As I discuss in Chapter 4, however, patterns (or parts of patterns) don't always look alike, even though they always contain the same information. In Figure 5-2, for example, the pattern shape is the same in all three systems, but it can be stretched, which shows that it isn't always identical whenever it's implemented or found.

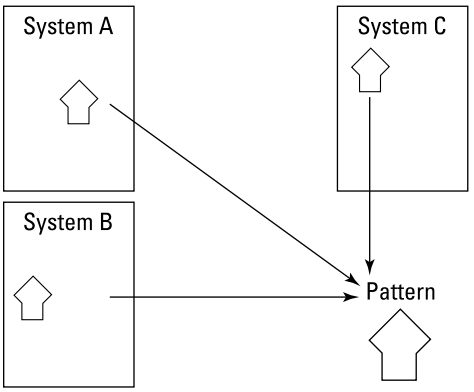


Figure 5-2: The Rule of Three: 1, 2, 3, pattern!

Extracting the general solution

After you determine that the solution has been used three or more times, the next step is extracting the general solution from the specific instances that you've observed. The solution should be as general as it can be but still specific enough to fit the observed instances.

After you write down the solution, you work backward to find the problem that was solved. This is sometimes hard because the problem may be subtle and not immediately obvious. It isn't useful to have a problem that asks, "How do I do *X?*" and a solution that says, "Do *X.*" This pairing doesn't help the reader understand the pattern.

If you find that you've described the problem, but you don't have the solution that you can make more general yet, stop now. There's a good chance that you'll be making the mistake of writing a pattern for some new idea. If you don't know the solution to a problem, you don't have something you can write as a pattern.

Writing the pattern document

After identifying the problem, think about what is required for the problem to exist in the observed form. This step is where things like programming languages and previously required patterns get written down.

Table 5-1 shows a template that you can use for writing your patterns, with the sections clearly labeled. Even experienced pattern writers find this template to be useful because it helps them to remember all the different kinds of information that is required.

Table 5-1 A Pattern Template

Pattern Section	*Section Contents*
Title	The title of the pattern. The title enters the design vocabulary element for the pattern.
Alias (or Also Known As)	This section lists alternative titles by which the pattern is known.
Context	The "Context" section describes where the problem exists. It lists the things that you can't change or that are assumed to be constraints on your system.
Problem	The "Problem" section has one or two sentences explaining the problem that's being solved. It should build on the "Context" section and should be concise.
Forces	The "Forces" section, which is the longest section, talks about the trade-offs involved with solving the problems. Unlike the "Context" section, the "Forces" section discusses things you can control and choose when applying the pattern to your design.
Solution	This section states the solution to the problem. It builds on the "Forces" section to explain why the solution is the right one and tells the reader what to do to solve the problem.
Sketch	This section of the pattern contains sketches or drawings to help the reader understand the solution.
Resulting Context	This section explains the state of the system space after the problem has been solved, including how the forces have been balanced. It discusses the context for any new problems that may have been introduced by this solution.

Pattern Section	*Section Contents*
Rationale	The "Rationale" section explains why this solution is the right one for the problem. It lists stories of successes with the solution or failures without the solution.
Related Patterns	This section contains patterns that are related to the current one. These patterns may be referenced in the "Context" or "Resulting Context" section, or they may be patterns that solve similar problems.
Author	This section lists the author's name and date. Patterns are frequently revised; the date tells the reader whether he or she has the latest version.

See Chapter 6 to see the similarities and differences between this list of sections and those in other pattern definitions.

Naming the pattern

The next step in pattern creation is coming up with a name for the pattern. Naming a pattern can be hard. Some names are fun and have inside meaning, but these are rarely the best names. You want a short, descriptive name that will easily fit into the lexicon of the pattern users.

During your search for a name, you may come up with several candidates, which are listed in the "Alias" or "Also Known As" section (refer to Table 5-1) of the pattern document. At this time, however, you want to label your pattern a *candidate pattern* so that your readers know that the pattern is still new and hasn't been reviewed yet (see the next section).

Although I encourage you to write patterns, I discourage you from labeling something a pattern if you aren't really sure that it is a proven technique. Doing otherwise may confuse someone else into using the unproven technique. After it has been reviewed, you can remove *candidate* from the name.

Getting expert reviews

The patterns that you find on websites or in books have been critically reviewed and revised three, four, or more times. The following sections describe some of the types of expert reviews that patterns typically receive. Reviews and workshops (discussed in the "Writers' workshops" section, later

in this chapter) have helped the pattern become succinct and understandable. Although not required of a pattern, I recommend getting feedback and having your patterns reviewed.

Figure 5-3 shows a typical progression of reviews, from review by peers to review by an editorial board before publication.

Figure 5-3: A pattern's journey of reviews.

PLoP conferences

Many published patterns have been though one of the many Pattern Languages of Programming (PLoP) conferences, which focus on new patterns that people are writing. For these conferences, the papers are shepherded — guided by experienced pattern writers — to make them as good as possible before review by a writers' workshop (described in the next section).

The North American pattern conference, PLoP, began at a University of Illinois conference center; lately, it has been moving around the country, usually co-located with another fun conference such as SPLASH or AGILE. Euro PLoP is the European pattern conference, which always happens in early July at Kloster Irsee near Munich, Germany. See `www.hillside.net/conferences` for descriptions of and information on other PLoP conferences.

In addition to these geographically based PLoPs, some special-purpose PLoPs are held occasionally, sometimes by invitation only. Some recent special-purpose PLoPs include Scrum PLoP for collecting the patterns of the Scrum agile method, Meta PLoP for metaprogramming patterns, and Para PLoP for parallel programming patterns.

Writers' workshops

Cultures are based on shared rituals, and the pattern community is built around the culture of a writers' workshop. In the same way that pattern authors are giving away their secrets and expertise when they write patterns, community members give their suggestions for improvements to the authors. One of the rules of a writers' workshop is that all comments must be suggestions for improvement. After the workshop, pattern authors are expected to take the feedback that they have received and to revise their patterns to make them even better.

All the participants in a writers' workshop are fellow pattern authors. This fact and the structured nature of the workshop contribute to it being a safe and respectful place to have your pattern reviewed.

Richard Gabriel is a Lisp programming-language pioneer and poet who brought writers' workshops to the pattern community. His book *Writers' Workshops and the Work of Making Things* (Pearson Education) explores the similarities and differences of these workshops in the worlds of software and poetry.

You can find a good guide to writers' workshops at `www.hillside.net/component/content/article/65-how-to-run-plop/235-how-to-hold-a-writers-workshop`.

You don't need a conference to organize a writers' workshop — you can hold one with your team or within your company.

Reading groups

In writers' workshops (see the preceding section), pattern authors get feedback to improve their patterns. If you want to understand a pattern in a group setting by discussing the merits and implementations, you can find a reading group. Reading groups exist in a number of cities to discuss pattern books or articles, or perhaps other computer topics, such as agile methods.

If you can't find a reading group, you can start one of your own.

You can find a good guide to reading groups at `www.industriallogic.com/papers/khdraft.pdf`.

Editorial reviews

If authors publish their patterns in a book or article, the patterns will be reviewed a few more times to ultimately make them the best, most understandable things that they can be.

Pattern-community principles

Throughout this book, I frequently mention the pattern community, which is based on the culture of reading, writing, and using patterns, as well as giving away practical experience in the form of patterns. Community members share several ethical principles:

- **Buschmann's Rule:** Never capture your own ideas in a pattern. To be sure that your good idea really is a pattern, let someone else write it. If you're writing patterns, write about the recurring solutions that someone else put into software.
- **Focus on broad, long-lasting, positive patterns:** Strive to make your patterns useful to many people.
- **Intellectual-currency paradox:** Ideas are worth more if they're given away. Pattern authors need to feel comfortable sharing the proven solutions that they've encountered and taken the time to write about. One way to achieve this is to give people credit for the patterns that they write and to keep the pattern-to-author association.
- **Encouragement and reward:** The community encourages its members to be secure about telling their secrets.
- **Reward:** People who created these techniques or who first took the trouble to commit them to writing are rewarded with ongoing credit for their pattern.
- **Aggressive disregard for originality (as put forth by Brian Foote):** The academic community strives for novelty, wanting new ideas and new results. The pattern community, on the other hand, seeks out the proven solutions. These were written about in nonpattern forms in many places in the past.
- **Proven solutions:** Patterns are solutions that have withstood the test of time. This leads to the Rule of Three (refer to "Confirming the Rule of Three," earlier in this chapter).
- **No hype:** Patterns are good for solving problems, but they won't solve all the world's problems or all your design problems.

Keeping patterns current

Individual patterns evolve and change over time, as do pattern collections and handbooks. New functionality is added, something is removed from the pattern because the tools now take care of it automatically, and so on. When you write your pattern and have it reviewed, you must be willing to remove, add, or change parts that the readers find necessary to understand the pattern. Revise your pattern continually!

Because the software world is changing so rapidly, you should try to write the fundamental patterns that have stood the test of time.

Pattern-Oriented Software Architecture: A System of Patterns, by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley), provides an example of the editing process. The Proxy pattern (see Chapter 19) was first published in *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional), with three variants. As the *Pattern-Oriented Software Architecture* authors revised that pattern, they realized that it had an additional four variants. After further review, including review at a PLoP conference, the pattern was published in *Pattern-Oriented Software Architecture: A System of Patterns,* with seven total variants. Technology continues to evolve; I've added an eighth variant in Chapter 19.

Documenting System Architecture with Patterns

Patterns explain a design. A professor of mine said, "Every problem in computer science boils down to trade-offs," which I've found to be true. Patterns capture those trade-offs so that you understand *why* the design is the way that it is.

Patterns work together to build a larger solution, as you see in Chapter 6. When patterns are used to design and build a system, the patterns are part of the system's documentation. This documentation explains the design choices, as well as how the parts of the solution fit together.

The project that you're working on may be the source of the patterns that you'll write about, as described in the previous section. If the design wasn't created with patterns, or if the project has some unique characteristics, you can use the pattern form to describe the architecture and convey to the readers of the patterns just where the "unmovable walls" exist. This refers to the aspects of the system that may look ordinary but that are actually important structural elements that are tying the whole architecture together. If you've seen the concepts implemented elsewhere, you can write a pattern.

Patterns make a natural complement to a software framework. Patterns can explain the components in the framework and explain where the framework user may need to add extra classes or configuration to make a working application.

A good example of using patterns to document the architecture of a system can be found in *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley). One case study describes a web

server that is intended to provide efficient caching and content delivery to Internet and intranet users. The authors identify seven common challenges that arise in developing concurrent servers such as this:

- Encapsulating low-level, operating-system APIs
- Decoupling event demultiplexing and connection management from protocol processing
- Using multithreading to scale up server performance
- Implementing a synchronized request queue
- Minimizing server threading overhead
- Effectively using asynchronous I/O
- Enhancing server configurability

Each of these problems is then examined in a short patternlike format that contains the following sections:

- **Problem:** States the problem and design choices that must be considered in this circumstance.
- **Context:** Explains where the problem exists within the overall design of the system.
- **Solution:** Cites the patterns that should be used to resolve the problem and balance the choices that the problem stated.
- **Use:** Explains how the patterns were used within the overall system design. This section usually includes a class diagram showing how the patterns fit into the overall system design.

Part II

Putting Patterns to Work

In this part . . .

To begin this part, I add to the definition of patterns by discussing how they're created, structured and categorized.

As you get more familiar with patterns, you'll want to remember where you found the most useful ones, so I also show you how to build a catalog of patterns — a personal reference handbook.

Finally, because just knowing where to find patterns isn't enough, I show you how to locate and then implement specific patterns to solve specific problems.

Chapter 6

Making Sense of Patterns

In This Chapter

- Classifying patterns
- Finding out about pattern collections and languages

Just as no two problems are exactly alike, no two patterns are identical. So, you'll find a wide range of patterns to solve a wide variety of problems. In this chapter, I tell you about some of the kinds of patterns that you'll find. These patterns have different scopes and scales, ranging from huge patterns that cover the big architectural styles described in Chapter 2 to small ones that help you get the most from your programming language.

You solve individual problems through the application of individual patterns. Most of the software design and architecture challenges that you confront are bigger and require several patterns to fully resolve them. To tackle this need for groupings of patterns, I talk about grouping patterns into collections and into a network of related patterns, which is called a *pattern language*.

Understanding Pattern Classifications

Patterns contain several required sections — including Problem, Context, Forces, and Solution — but there are many ways to combine these sections. If you've looked at any patterns, you probably noticed that they all appear slightly different, and you may have even thought that some of them weren't patterns at all!

I explain the parts of a pattern and the contents of its parts in Chapter 4.

There are many ways to slice up the universe of patterns. So, in this section, I describe the most common classification schemes, starting with the pattern styles you're most likely to encounter.

Styles

If you open a poetry anthology, you'll see that the poems are written in many different styles. Patterns are similar. Pattern authors don't always choose to write their patterns in the same styles, so they vary the way that the material is presented. *Pattern styles* are the different ways of combining the same essential sections into a pattern.

Table 6-1 compares styles from several sources:

- This book
- *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley)
- *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional)
- *A Pattern Language,* by Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel (Cambridge University Press)

As you can see in Table 6-1, all the pattern styles contain the same information, even though they display it in different sections with different headings. Focus your attention on the left two columns, which describe the sections of a pattern and how patterns appear in this book.

Table 6-1 Pattern Style Comparison

Section	*This Book*	*Pattern-Oriented Software Architecture*	*Design Patterns*	*A Pattern Language*
Context	Example problem or pattern introduction	Context	Motivation	Normal text at the beginning
Problem	Example problem or pattern introduction	Problem	Motivation, intent	First bold text
Forces	Example problem or pattern introduction	Problem	Motivation	Between bold text

Section	*This Book*	*Pattern-Oriented Software Architecture*	*Design Patterns*	*A Pattern Language*
Solution	Solution section or before the implementation section	Solution, structure, dynamics, implementation	Applicability, structure, participants, collaborations, implementation	After *therefore*
Resulting context	Exploring the impacts	Solution, consequences	Consequences	After the second row of three stars
Consequences	Exploring the impacts	Consequences	Consequences	After the second row of three stars
Rationale	Various places, including example problem, pattern introduction, or solution description	Example resolved	Motivation	Between bold text, after the second row of three stars
Known uses	Not called out in a separate section	Known uses	Known uses	Between bold text, after the second row of three stars

There are several reasons why pattern authors use these different styles. In some cases, they think that one style will be more understandable to their target audience; in other cases, they think that one section or another provides the most important information. The differences add to your confusion as a pattern reader, but when you dissect patterns, you can see that all the different styles contain the same information.

As you're building something with the aid of a pattern, it's important to know where to look to find the information that you need. It'll be in there somewhere!

Depth

Not all patterns have the same *depth* — meaning that they don't address the same scopes or scales of problems (see Figure 6-1). Some patterns are written at a very low level, such as:

> Use a left or right bit shift operation to multiply or divide by a power of 2.

This pattern describes a particular way of solving a math problem if the computer doesn't have built-in multiplication or division functions (which is still sometimes the case!).

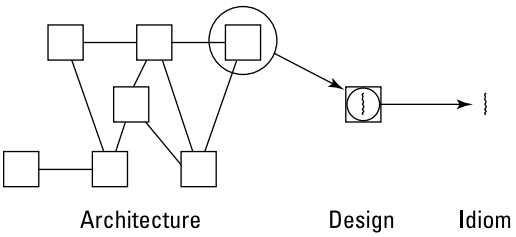


Figure 6-1: Patterns at various levels.

Other patterns are high-level, describing the structure of an entire system.

Model-View-Controller (MVC; see Chapter 13) is one of these patterns. It describes the overall solution instead of solving the problem of implementing a tiny detail.

As your design progresses from a blank piece of paper or blank window in your development environment to a fully fleshed-out project, you'll be able to apply patterns for each of these scopes. You'll start with a pattern to introduce the structural architecture into the system. Then you'll move to solving the problems related to just a few classes or components. Finally, you'll need the low-level patterns to help you actually get the most out of the language or software/hardware infrastructure. If you're just looking for the solution to a problem that you encounter when working with just a few components, you may be able to jump directly to the design patterns and ignore the architectural patterns.

The different classifications that I describe in the next few sections help you find a pattern of the correct scope to solve your problem.

Architectural

Architectural patterns define the structure of the solution at the highest level. These patterns influence the entire system. The patterns in Part III of this book, for example, are all architectural patterns because they set the direction and overall structure of the system.

Architectural patterns require all the parts of the system that are touched by the pattern to participate in its design. An example is my Minimize Human Intervention pattern, which helps make computing systems more reliable by automatically performing as much action as possible and only rarely engaging humans to help. All parts of the system must participate to receive the pattern's benefits. If one part of the system doesn't participate in implementing the

architectural pattern and asks a human to "Click OK to continue," that part will make system reliability lower and prevent the computing system from achieving its availability goal.

Because architectural patterns affect the entire system, they need to be included in the design first. You'll run into problems later if you get deep into your design and then decide to structure it by using one of the architectural patterns, because you'll already have built parts of the system that don't conform to the architectural pattern.

Architectural patterns can solve problems related to the functional requirements discussed in Chapter 1. Sometimes, an architectural pattern structures the system or adds an element to the system that resolves a functional requirement. For example, the Layers pattern (see Chapter 9) divides the system into stacking blocks of functionality. This pattern implements the requirement that the solution be modular and that the parts of the solution be easily interchangeable.

Architectural patterns also add nonfunctional capabilities to the system. The Minimize Human Intervention architectural pattern, which I mention earlier in this section, supports increased availability, which is a nonfunctional requirement.

Architectural patterns don't specify the whole structure down to the lowest level; they describe high-level problems and solutions. They're written at a high level so that they can be applied in a variety of circumstances.

When you select and use a specific architectural pattern, you're choosing to apply that pattern's architectural style to the system. (See Chapter 2 for an introduction to architectural styles.) Following are patterns covered in this book that you can use to design four different architectural styles:

- **From mud to structure:** Layers (Chapter 9), Pipes and Filters (Chapter 10), Blackboard (Chapter 11)
- **Distributed systems:** Broker (Chapter 12)
- **Interactive systems:** MVC (Chapter 13), Presentation-Abstraction-Control (Chapter 14)
- **Adaptable systems:** Microkernel (Chapter 15), Reflection (Chapter 16)

Architectural patterns usually point you toward the design problems you need to solve through their Resulting Context section (described in Chapter 4). This section of the pattern gets you started by pointing out the next problems that you need to solve to fully implement the system. With these problems, you use design patterns and idioms, which I discuss in the next two sections.

Design

Design patterns solve individual design problems within a particular context. In the pattern scope hierarchy, design patterns are the medium-scale patterns. They address problems across a system, but instead of involving the entire system, as architectural patterns do (see the preceding section), they affect only a few components. They help you solve individual design problems in building components or small groups of components.

You use a design pattern to fill in gaps in the architectural style that you chose with your architectural pattern. When you select an architectural pattern, you'll frequently find that there are some loose ends. MVC (Chapter 13), for example, introduces a tight coupling between the models and the views and controllers. This coupling can become a problem if there are many views and controllers, so you have a design problem that needs to be solved. The design pattern Command Processor, discussed in Chapter 20, addresses this problem.

The best-known source of design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software.* Its 23 patterns address common problems in object-oriented design and help you build robust sets of components easily. These may be the only patterns you've run into before this book, so I want to show you that you can use many other patterns for all kinds of problems — not just design problems related to classes and objects.

Within the general scope of design patterns, patterns can be classified further. The *Design Patterns* authors use three sub-classifications: Creational, Behavioral, and Structural (see "Other classifications," later in this chapter). In this book, however, I use the following sub-classifications:

- Structural Decomposition
- Organization of Work
- Access Control
- Management
- Communications

You can find patterns in all these sub-classifications in Part IV of this book.

Idiom

The lowest-level patterns are *idioms.* Idioms have the narrowest scope and are specific to a programming language or platform, but they help you get past their limitations. The example in the "Depth" section, earlier in this chapter, about bit-shifting left to multiply by powers of 2 if the computer doesn't have a multiplication function, is one example of an idiom. Other examples of idioms are patterns that help you overcome the limitations of different memory-management methods employed in different programming languages.

Chapter 22 contains one idiom, Counted Pointer, that's widely used in many languages even though it was designed for C++. This shows that multiple versions of the same idiom may address the unique requirements of different programming languages.

Idioms contain language-specific guidance. When you start looking for an idiom, you may find one for a different language from the one you're working in. Don't immediately discard it. You may be able to extrapolate its solution into your design or code, or it may give you a new way to look at your problem that will lead you to the right idiom for your language.

You can learn something from every idiom that's related to your problem, even if it doesn't solve that problem.

Other classifications

Just as there are multiple pattern styles, or ways that patterns are written, there are multiple ways to classify and group patterns. The classification system of architectural and design patterns and idioms described in the previous sections has been widely used for many years. Other pattern authors focus on particular aspects of the system. *Design Patterns: Elements of Reusable Object-Oriented Software,* for example, divides design patterns into three classifications:

- **Creational:** Creational patterns such as Factory Method and Abstract Factory describe ways to create new objects.
- **Structural:** Structural patterns such as Composite talk about how to structure the system by combining objects in beneficial ways.
- **Behavioral:** Behavioral patterns such as State and Memento cover the patterns that address the runtime dynamics of the system.

A system by Doug Schmidt is close to my method of Architectural, Design, and Idiom but groups them into only two categories: Strategic and Tactical. Strategic patterns are similar to the architectural patterns introduced here. They define the overall structure of the system and how the major components should interact; they describe the strategic decisions that affect the entire system's design.

The design patterns are like Schmidt's Tactical patterns. They address specific design problems. They don't address the grand scale of the system; instead, they help with the individual problems encountered in design. You can see how my system interacts with the Schmidt system in Figure 6-2.

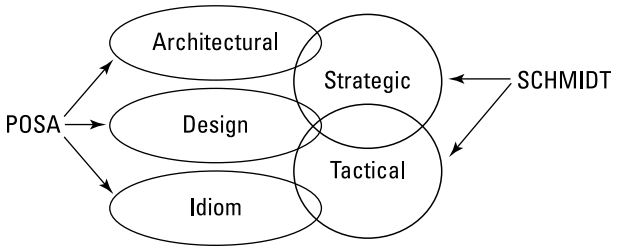


Figure 6-2: Different categories of patterns.

All the classifications are useful in their own ways. You'll want to collect a set of patterns across the different classifications and put them in your toolbox. Chapter 7 tells you how to create a catalog of patterns that you can refer to frequently.

Grouping Patterns

The pattern classifications define categories of patterns that are useful to you at different times while building your system. In this section, I introduce two different ways of grouping patterns based on the domain of technology the patterns cover rather than the scope of problems they solve.

As I mention in the discussion of design patterns earlier in this chapter, patterns address and resolve the problems that are left unanswered by other patterns. Sometimes, several patterns can be used together to solve a problem that is much bigger than any of the patterns being used. Patterns work together in these ways, as I tell you in the upcoming section about pattern languages.

Not all the patterns within a collection or a language are meant to be used to solve any given problem. Both groupings contain more patterns than you'll actually use to solve any practical design problem.

Pattern collections

Put any group of items together, and you can call the group a collection. Many of the published sets of patterns out in the world are just that — collections of patterns. They're grouped for convenience. Maybe they fill out a book or were written by the same author.

Patterns in a collection can work together to solve bigger problems, such as when you use both Command and Memento (from *Design Patterns: Elements of Reusable Object-Oriented Software*) to implement an undo function. More often than not, though, pattern collections are just catalogs of patterns that address similar problems.

Collections of patterns are good, but a collection may not contain patterns that address every aspect of your software problem, which can leave you hunting elsewhere for other patterns. Collections usually don't give you guidance on how to use the patterns together in a system either. Your search may take you to pattern languages, which I discuss in the next section.

Pattern languages

A *pattern language* is another grouping of patterns that work together to solve a bigger problem. Unlike mere collections, however, languages provide complete coverage of a problem space.

The term *pattern language* has a specific meaning and is used carefully within the community of pattern authors. You won't need to spend any effort figuring out whether something is a pattern collection or language; the writers of pattern languages are proud that they've put together a language, and they'll point out that fact to you.

Languages need to be complete enough to address all the problems found within them, leaving no loose ends. So a language that explains how to log status messages using a framework like `log4j` can't leave a gap. If it doesn't, it isn't complete. It must address the problem of log files reaching their maximum size, for example.

Pattern languages also need to be complete in that they solve enough problems to build something. A collection may be complete, but it may not design a useful, whole solution. A pattern collection for logging that addresses the need for timestamps and keeps messages in order is useful, but if it doesn't include the basics to help you with when, where, and what to log, it doesn't design a complete solution and it won't be considered a language.

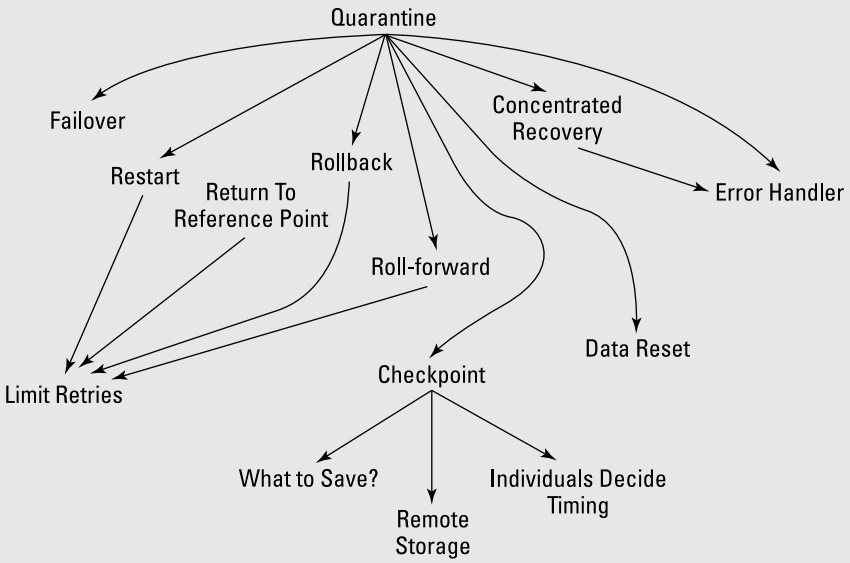
Pattern languages exhibit one of the enabling techniques that I described in Chapter 2 — languages are sufficient and complete.

The patterns within a pattern language can be combined in many ways. The pattern-language writers will probably offer some guidance on how the patterns relate (see the nearby "Reading pattern languages" sidebar). Their ordering is just a suggestion, however, and you can apply the patterns in any order or sequence your problem needs.

Reading pattern languages

There isn't complete agreement in the pattern community today about how pattern languages should be documented, so you'll see some variety. This variety just indicates that the community has many minds all trying to write the best patterns that they can, as you see in different pattern styles and in different pattern-language styles. Here's what I think are the most important elements in a pattern language:

- **Title:** Pattern languages need titles, just as individual patterns do.
- **Context:** This section explains when and why a pattern language should be applied. It is similar to the context within an individual pattern. The language context also explains, in words, the relationships between the patterns; it links the resulting context of one pattern with the context of the next pattern. (See Chapter 4 for more information on these sections.)
- **Map:** A pattern language has a map or language diagram to show one way that the patterns connect and can be related to one another. This map graphically shows the relationship of one pattern resolving the problems introduced by another pattern. The following figure shows an example map for a language that addresses software error recovery.
- **The actual patterns:** A pattern language is ultimately made up of individual patterns. Either the actual patterns or references to those patterns should be part of the language.



Reproduced with permission of Alcatel-Lucent: Patterns for Fault Tolerant Software, 2007, Hanmer. A John Wiley & Sons publication.

Chapter 7

Building Your Own Pattern Catalog

In This Chapter

- Creating a personal catalog of patterns
- Updating your pattern catalog

When you start using patterns to solve your design problems, you'll find that you have a few favorites — the ones that you know inside and out and refer to frequently to solve the kinds of problems that you usually confront in your software development.

To make your work easier, you should record these favorite patterns in your own pattern catalog for future reference. You can turn to this pattern catalog when you start a new design problem to get an idea of how to structure your solution. You also can turn back to it to follow the implementation steps that tell you how to incorporate the pattern's solution in your design. That way, you don't have to memorize all the details of all the forces or all the implementation steps, because the catalog holds all that information.

The patterns in your personal catalog constitute a collection and include pattern languages or parts of languages that are useful to you. It's your own personal "software handbook." You may include patterns from different pattern languages — one for interactive system design, one for logging, and another for user interface design, for example — as well as patterns that aren't included in complete languages, like the ones in *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).

Your catalog also should include well-known patterns like the ones I list in Chapter 23. And, of course, I hope that the patterns in this book will become your favorites and will be part of your catalog.

Assembling Your Catalog

In this section, I give you some pointers on putting your pattern catalog together.

This is *your* catalog, not the one and only pattern catalog for everyone in the world, so customize it to make it truly your own. Record information in your own words in a place where you'll remember it. Add to it as your needs and interests change. Refer other people to the patterns in your catalog when they ask how you solved that really difficult problem so effortlessly.

Choosing a medium

Your pattern catalog can be in whatever medium works best for you, as shown in Figure 7-1.

Make your choice of medium based on how you use patterns:

- If you take your patterns to meetings to discuss them with your colleagues, index cards work well.
- If you refer to your catalog only when you're near your computer or the web, a personal website or wiki works great because you can add hyperlinks.
- If most of your references are already on paper, a loose-leaf notebook may be the best medium for you.

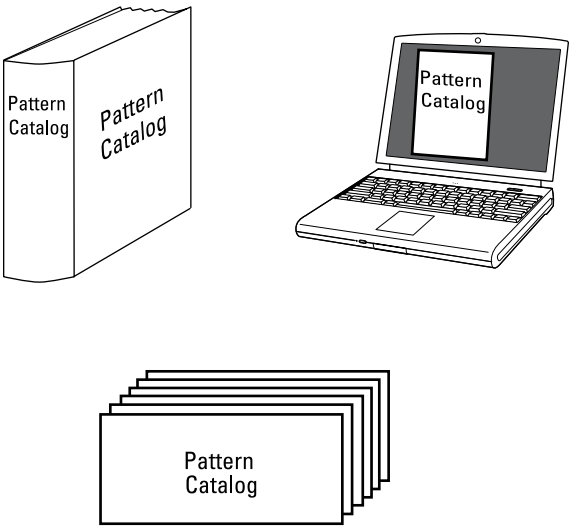


Figure 7-1: A software pattern catalog can take many forms.

The most important point is to make your catalog convenient for you so that you don't forget to refer to it.

Identifying the problems you face

Whether you're writing software at a company, in a class, or for fun, you're probably focusing your efforts within a certain area. Within that area, you're going to find many repeated problems and many of the same situations.

For example, you may always have to deal with a user interface, or with JavaScript, or a program in a small memory system. These situations may not be problems for you in the sense that they're difficult for you to solve; they're just the types of software projects that you find yourself working on repeatedly.

As you develop your software, take notes about the general areas of problems that you encounter, paying special attention to the following:

- The topics on which you continually ask experts for guidance
- The topics that you've bookmarked in your favorite reference books or on your favorite websites
- The topics that you and your friends discuss repeatedly without agreeing on a resolution
- The topics that your colleagues or friends come to you for help with

Finding patterns that solve your problems

After writing down the problem areas, look for patterns within these areas. You can find them in tons of books and on hundreds of websites about patterns, so your search for patterns to include in your catalog isn't going to be done in one sitting.

Here are a few sources for patterns of various types:

- For basic object-oriented design, look to the *Design Patterns* patterns.
- For architecture-level patterns, refer to Part III of this book.
- For user-interface patterns, turn to the Interaction Design Pattern Library at `www.welie.com/patterns`.

As you go about your regular software work, make notes about where you find patterns. While you look through these collections, record a few facts about each pattern that interests you, such as the problem, context, and

solution, as shown in Figure 7-2. The most important thing to note is where to find the pattern again.

To get you started with your catalog, Chapter 23 lists ten patterns that everyone should know, and Chapter 24 contains the top ten places (other than this book) where you should look for patterns.

Finding patterns to solve your problems will be a lifelong (or at least a career-long) pursuit — it isn't a one-time effort.

Organizing the catalog in sections

When you've found a bunch of patterns within your problem areas, you should label them to identify where they should be if your catalog were sorted different ways.

You'll find it most useful to organize your catalog into two main sections:

- **Pattern categories,** such as architectural, design, and idiom
- **Problem categories,** such as enterprise, database, and user-interface patterns

Table 7-1 contains some example pattern labels for your catalog. As you can see, you'll have more problem categories than pattern categories. The reason? Problem categories span the universe of problems in computing, which is naturally larger than the universe of solutions.

ARCHITECTURE
  Interactive Systems

Flexible HCI	→	Model-View-Controller	POSA
Agents	→	Presentation Abstraction Control	POSA

DESIGN
  Classes

Decoupling Abstraction	→	Bridge	GOF
Adding Responsibility	→	Decorator	GOF

Checkpointing	•	Patterns for Fault-Tolerant Software
	•	
	•	

Figure 7-2: Pattern notes.

Table 7-1 Example Catalog Sections

Pattern Categories	*Problem Categories*
Architectural Patterns	Business Computation
Design Patterns	Business Process
Language-Specific Idioms	Client-Server
Computer-Specific Idioms	Communications
Organizational Patterns	Concurrent Systems
Process Patterns	Database
Analysis Patterns	Distributed Systems
	Event-Driven Systems
	Fault-Tolerant Systems
	GUI Development
	Interactive Systems
	Memory Management
	Multimedia
	Networking
	Parallel Programming
	Pattern Writing and Reviewing
	Real-Time Systems
	Refactoring
	Security
	System Modeling
	Testing
	Training
	Transaction Processing
	Website Development

These categories will help you choose patterns from your catalog to solve your design problems (see Chapter 8).

Connecting the patterns

As you add patterns to your catalog, think about how they relate to the other patterns already in the catalog. Patterns are rarely used by themselves. Much more often, they work together to solve bigger problems.

As you find patterns, write down the connections among them. Add this information to an electronic catalog as a link or to a paper catalog as a note in the margin. These connections can help you remember things that make sense to you about groups of patterns. You may enter something like "When I use Riding Over Transients, I also use Leaky Bucket Counter."

Here are a few attributes that may lead you to connect two patterns:

- The patterns are usually used together.
- The patterns are complementary, so you may use one or the other but not both.
- The patterns are similar but used in different contexts (one in Java and the other in C#, for example).
- The patterns come from a hard-to-remember source, and noting this relationship can help jog your memory about the source.

Keeping Your Catalog Current

Despite your best efforts, you'll always have gaps or outdated material in your pattern catalog for a variety of reasons, such as these:

- Well-known solutions aren't always written as patterns.
- A recurring problem may not have a solution, because whenever it appears, the forces are always quite a bit different.
- New patterns are being published all the time, and you don't have time to read them all.
- Published patterns are being revised continually, slowly in books or quickly on the web, but perhaps faster than you're able to keep up with the changes.

All these factors contribute to the risk that your catalog will grow stale over time. Don't worry — just keep the best catalog you can, and maintain it as often as you can.

Review your catalog periodically, adding helpful new patterns and removing the patterns that are no longer useful. You may need to remove patterns from your collection for a variety of reasons, including the following:

- The problem has disappeared.
- Better alternatives (pattern or nonpattern) are available.
- Technology has evolved.

You may want to keep some outdated patterns in your catalog, however. Even if you won't use them in new designs, you may want to refer to them as you maintain older systems that used those patterns. As you use patterns from your catalog, make notes to remind yourself where and when you used them.

Chapter 8 tells you how to choose a pattern from your catalog, or from all the available patterns, to solve your design problem.

Chapter 8

Choosing a Pattern

In This Chapter

- Evaluating patterns like a critic
- Selecting the pattern you need
- Using patterns to solve architectural problems

You can find lots and lots of patterns, because many people have found patterns to be useful ways to document proven solutions to problems. Because so many patterns are out there, you have to read a pattern carefully to decide whether it's relevant to your problem. In this chapter, I tell you how to make that decision.

This chapter starts with features you can use to evaluate patterns. Then I outline seven steps for choosing a pattern that solves your particular problem. When you're comfortable selecting patterns, you can create your system's architecture with patterns, as I explain at the end of this chapter.

Examining Patterns Critically

Using a published pattern is like getting advice. If the advice is from a total stranger, you'll want to check to see that the advice is good. If the advice is from your best friend or someone whose expertise you trust, you're more likely to follow the advice, but you'll still consider whether the advice is appropriate for your situation.

Similarly, not every pattern will be right for you. You need to determine whether a pattern will help you solve your problem. And, in order to do this, you need to read the pattern with a critical eye. In this section, I show you how to do exactly that.

Asking the right questions about patterns

The first step in evaluating a pattern is to ask yourself a few key questions about the pattern. Armed with the answers to the following questions, you can begin to get a sense of whether the pattern is right for you:

- **Is the pattern useful?** In other words, does the pattern solve a real problem and, in particular, does it solve the problem that you currently have? The pattern may be useful for another problem, but if it doesn't solve *your* problem, it won't be much help.
- **Does the pattern contain enough information to implement the solution?** Sometimes you're looking for a pattern that will help you with the overall structure of the system, but you won't be implementing anything yet. Other times, you'll want detailed implementation instructions because you need to write some code now. The pattern should provide the kind of implementation assistance that you're looking for.
- **Does the author know something about the field and topic?** This question is sometimes hard to answer, but if you know that the author is an expert in the field in general and the topic in particular, then the pattern will probably be more useful than a similar pattern written by someone who *isn't* an expert in the field and topic.

Knowing what to look for in a pattern

Looking for the right pattern for your particular situation is a lot like looking for a mate. All kinds of great patterns are out there, but you don't want to settle down with the first one you find. Instead, you want to find a pattern that has certain traits. Here are some key traits to look for in a pattern:

- **The problem statement is clear.** If you don't understand what problem the pattern is solving, it probably isn't applicable to your situation.
- **The pattern has an appropriate scope for the problem.** The scope tells you when in the design process you'll apply the pattern.

  There are three categories of patterns (see Chapter 5):

  - **Architectural patterns:** You need an architectural pattern if you're just putting the big building blocks together.
  - **Design patterns:** You need a design pattern if you have specific design problems to solve.
  - **Idioms:** You need an idiom if you're trying to work through a language or some problematic nitty-gritty detail.

Not all patterns tell you their scope, so you'll have to determine the scope yourself. To know what scope is most useful, check in the solution and structure sections to determine if it involves widespread parts of the system or if it's all related to a single, small part of the system. Widespread interaction places it toward the architectural patterns and small interaction or small problems give it the scope of an idiom or design pattern.

- **The pattern actually solves the stated problem.** Sometimes the solution doesn't really address the problem it says it addresses. This is a sign of an immature pattern, which you sometimes come across in searching for patterns.
- **The pattern's context matches the problem's context.** You're solving a problem, and that problem exists in the context of the rest of the system. Make sure that the pattern's context matches yours, with the right language and the right kind of system. Ideally, the pattern mentions that it solves the problem you have.

  If the pattern you're looking at is an idiom, pay special attention to whether its context matches yours. If the context doesn't match, see whether the pattern offers some advice about your situation anyway.

Don't disregard a pattern just because it doesn't say it's for your specific context. If you're looking for a design pattern to solve a particular problem, but instead you find an architectural pattern, think about what the pattern tells you about how the design should be built. It may contain enough information to solve your problem. If you're writing in Java and you find a C++ idiom, think about how that C++ idiom still applies to your problem and then adapt it, or pull some keywords from the idiom to help refine your search.

- **The pattern contains enough information for you to implement it.** You want to build something with a pattern. Does the pattern give you enough information that you could easily include the pattern's solution in your design or architecture?
- **The pattern fits with what you've already designed.** Patterns can work together very well to build a solution. Make sure that the patterns you apply build toward the solution. If the pattern seems to have you backing up in your design, resolving problems you've already addressed, it may not be the right pattern for you. On the other hand, a pattern may point out a flaw in what you've already designed.

Selecting a Particular Pattern

Now that you know how to evaluate patterns (see the previous section), you're ready to start searching for and selecting patterns. You can find patterns in many places, such as in this book, in other books of collected

patterns, in magazines such as *IEEE Software* and *Dr. Dobbs,* in technical conference papers, and on the Internet. As you develop your personal pattern catalog (see Chapter 7), you'll discover sources for the patterns that are most relevant to your work.

Figure 8-1 shows a seven-step method for selecting a pattern for use in a design. I explain this method in detail in the following sections.

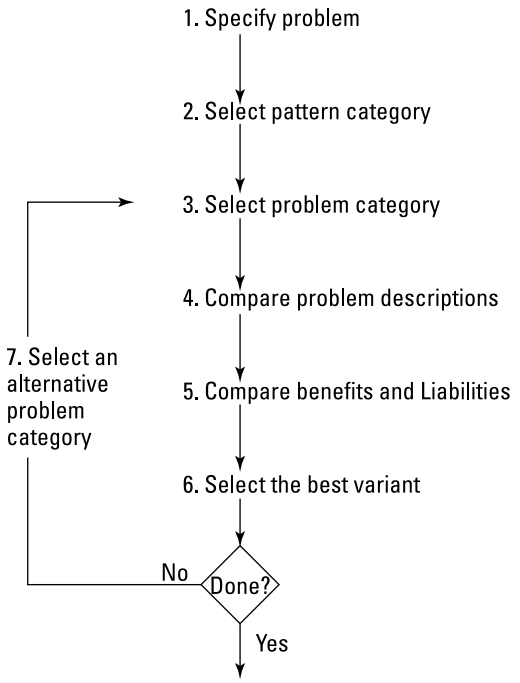


Figure 8-1: Seven steps to selecting a pattern.

Step 1: Specify the problem

The first step in finding a pattern is identifying the problem that you want to solve. The problem needs to be concrete like "securing a three-tier architecture," rather than something general like "making a website."

Splitting your problem into sub-problems

If the problem seems to have several parts, split the problem into smaller sub-problems, such as "managing the user interface" and "setting up the back-end database." Sometimes, the sub-problems aren't related to the structure but are related to the nonfunctional requirements — for example, if you need to design a web service that behaves in a certain way and the solution needs to be highly available. Divide this problem into two sub-problems.

Finding a pattern to address the small problems is easier than finding a single pattern that solves the larger, more-complex compound problem. Sometimes you get lucky and find the pattern for the larger problem — so before splitting your problem up, make a quick check for that larger problem's pattern.

Finding a context

For each sub-problem that you've identified, think about the constraints on the problem. These constraints make up the *context*.

For example, consider laying out the basic structure of an interactive text editor. A requirement of the system is that it should be able to adapt to different user-interface libraries and different style guides from your customers. These requirements define the context of the problem.

Considering trade-offs

After considering the context, think about the things that you have to balance to achieve a good solution. These are the trade-offs, the things that you have some flexibility about and can make choices about. What are some of the trade-offs in each sub-problem? For example, adding fault tolerance to a system increases its reliability and availability trading off against higher code complexity, longer development time, and more code to execute — which lowers performance.

The trade-offs may not be immediately obvious to you, but as you start reading patterns, you'll start seeing the trade-offs.

Consulting your pattern catalog

Reflect on the patterns in your catalog (see Chapter 7), and consider whether any of them is a good match for the problem and trade-offs that you've identified. Your pattern catalog comes into play here because it contains your most useful and most often-used patterns.

Step 2: Select the pattern category

After you've identified the problem for each sub-problem, you need to decide which pattern category will address the one you're solving. The pattern category is related to how patterns are indexed and grouped:

- **If you're still defining the basic structure of the system,** you should look for an architectural pattern (see Part III).
- **If you're structuring a few components of an architecture,** look for a design pattern (see Part IV).
- **If you're implementing something in a specific programming language,** look for a language-specific idiom (Chapter 22 has an example).

Table 8-1 lists the patterns discussed in Parts III and IV.

Table 8-1 Pattern Categories

Architectural Patterns	*Design Patterns*	*Idioms*
Layers	Whole-Part	Counted Pointer
Pipes and Filters	Master-Slave	
Blackboard	Proxy	
Broker	Command Processor	
Model-View-Controller	View Handler	
Presentation-Abstraction-Control	Forwarder-Receiver	
MicroKernel	Client-Dispatcher-Server	
Reflection	Publisher-Subscriber	

The goal of this step is to begin narrowing the number of patterns you need to review to find the best pattern to solve your problem.

For the example I'm using in this chapter (see "Finding a context," earlier in this chapter), because you're defining the basic structure of the interactive text editor, you need an architectural pattern.

Step 3: Select the problem category

After you identify the pattern category that will provide your solution, you need to identify the category of the problem that you want to solve. Within a collection of architectural patterns, for example, you see patterns in several problem categories, sometimes called *domains*.

The problem category helps you narrow your search for patterns that solve your problem. For example, you wouldn't look for a solution to a problem of interactive systems in a collection of patterns that focuses on inter-process messaging.

Table 8-2 shows some example problem categories.

Table 8-2 Problem Categories

Architectural Pattern Problem Categories	*Design Pattern Problem Categories*	*Idiom Problem Categories*
Structure	Structural	Java
Distributed system	Creational	C++
Interactive system	Behavioral	Small memory systems
Adaptable system		Ruby
Real-time system		Smalltalk
Fault-tolerant system		Performance tuning

Using the example of a search for an interactive text editor, you may look for patterns in one of the architecture styles introduced in Part III of this book — in particular, the two patterns in the interactive system problem category: Model-View-Controller (MVC; see Chapter 13) and Presentation-Abstraction-Control (PAC; see Chapter 14).

If you can't find a pattern that matches your problem perfectly, select a different problem category.

Step 4: Compare the problem descriptions

Now it's time to browse the collection for patterns that can address your specific problem. In this step, you'll use the detailed knowledge of the problem from Step 1 to narrow your search to just a couple patterns.

Use your detailed problem knowledge to look at the problems from the candidate patterns, as follows:

1. **Determine whether the candidate pattern's problem matches the problem that you're trying to solve, either completely or partly.**
2. **Determine what other patterns the candidate pattern requires you to have applied.**

   If you haven't already built in the patterns that the candidate pattern expects, consider whether you can — and whether doing so would make your software better or worse.
3. **Check the structure of the candidate pattern against the structure of your problem.**

Ask yourself whether the author broke the overarching problem into different sub-problems from the ones you created and, if so, whether you went too far in creating sub-problems. The answers to these questions may lead you to redefine your problem in a way that matches the pattern better.

If a promising candidate pattern solves a problem that's bigger than the narrow problem you're seeking help with, keep it on the candidate list because you may need it later.

4. **Check the candidate pattern against your context.**

   A pattern that doesn't match should be removed from the candidate list. But as you cross it off, think about whether it mentions something that you've forgotten.

The architectural pattern category and interactive systems problem category (see Part III) contains two patterns — MVC and PAC — that may be useful in the interactive text editor example. Both patterns support the requirement to change the user-interface style (refer to "Finding a context," earlier in this chapter). Because the final solution is expected to be tightly coupled rather than implemented through distributed agents, MVC will be more appropriate.

If you find that none of the patterns you're considering matches the problem you want to solve, maybe you're trying to solve the wrong problem. Start over with Step 1 using what you've learned the problem *isn't*.

Step 5: Compare benefits and liabilities

In this step, you look at the patterns you think should be considered further, paying particular attention to the trade-offs and consequences of applying them.

Trade-offs are the things that you can control in your design. Maybe you can trade flexibility for better performance or structural complexity for easier access to the back-end database. See whether making the trade-offs needed by the pattern solves your problem and meets the needs of your application. If the trade-offs result in missing your requirements — or if the trade-offs defeat the point of your application — remove the pattern from consideration.

If you're designing a system that you expect to be used for a long time, with many updates, but the trade-offs in the pattern you're considering value one-time efforts over maintainability, stop considering the pattern.

Patterns also contain a discussion of the consequences associated with applying the solution. These consequences are the benefits that happen when you apply the solution. The ease of adding new views to an interactive text editor is a benefit you'd receive after applying the MVC pattern, for example.

Sometimes, patterns have negative consequences, or *liabilities,* associated with the solution. In many cases, the liabilities associated with a solution are new problems that another pattern can solve.

Read through the consequences of the patterns you're considering. Do they give you the benefits that you were hoping for? Are the liabilities manageable, in other words can you resolve them easily either through applying another pattern or by a little design work on your part? Does application of the pattern provide a clear path toward a design or does it introduce more new problems than it solves? Does it have negative consequences for the overall design (for example, duplicating components already present in the design)?

After you consider the benefits and liabilities, choose a pattern that provides the benefits you need and that doesn't introduce liabilities you can't manage.

Figure 8-2 shows the process of narrowing down the large universe of patterns to one specific pattern.

If Steps 2 through 5 don't work and you haven't found a pattern that's appropriate for solving your problem, you'll have to develop a new solution to the problem.

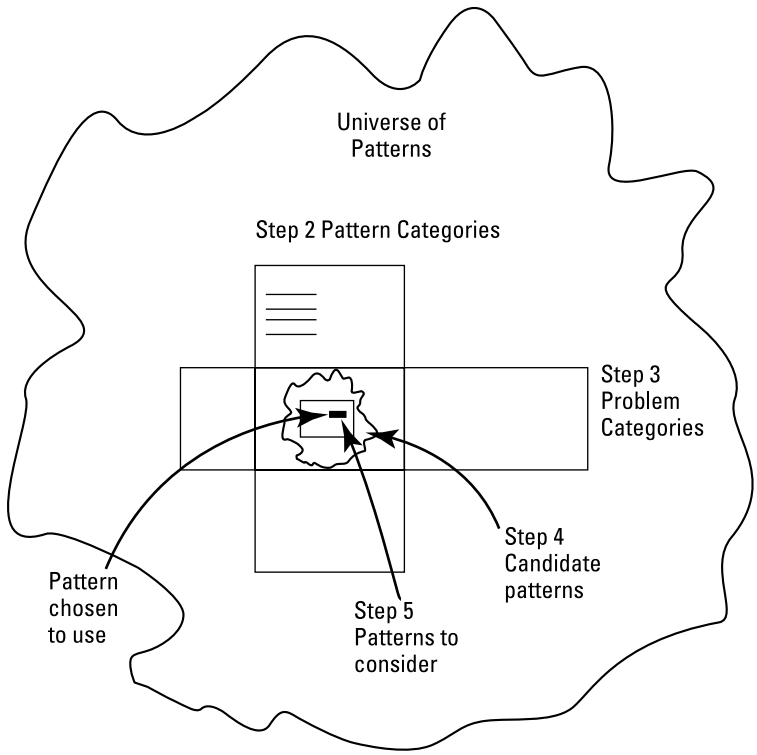


Figure 8-2: Narrowing the search to find the right pattern.

Step 6: Select the best variant

Patterns sometimes contain variants that offer alternative ways to implement the solution. MVC, for example, contains a Document-View variant that provides a different way of implementing the solution by relaxing the boundary between the view and the controller. In some cases, Document-View is a better solution than the main MVC solution. This is the case with the example interactive text editor I've been walking you through, for which you'd pick the Document-View variant, because the view and the controller are tightly interwoven.

If the pattern that you've identified has variants, in this step you decide which variant to apply.

Read the pattern's variants to determine how they differ from the main solution and whether any of them matches your circumstances better than the main solution does.

The variant's discussion sometimes explains how to implement the variant. More likely, though, you'll have to map the changes between the variant and the main solution to the implementation steps yourself.

If you've found a pattern and identified a variant that solves your problem or sub-problem, you can continue your design. If not, continue to the final step.

Step 7: Select an alternative problem category

If you couldn't find a pattern that met all your needs and solved your problems, try your search again, but this time, broaden your problem category. Instead of looking for a solution for a problem with your three-tier architecture, for example, look for a solution for the bigger enterprise system problem that you have.

Something else you can try is to look for closely related patterns that can guide you by giving you a little insight into what may work or won't work (refer to "Knowing what to look for in a pattern," earlier in this chapter).

Many patterns are specializations of patterns in other categories. The Composite pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional), for example, is a general design pattern. Many people have found it useful to publish problem-category-specific variations on the Composite pattern that fit the unique situations of their own problem domains.

If you find one of these specialized patterns that's similar to what you need, check out the general pattern on which it's modeled; that pattern may provide the design solution you need.

Designing Solution Architecture with Patterns

This section moves from selecting individual patterns to designing and implementing your system with the guidance of patterns. Patterns complement existing design methods by identifying solutions to individual problems. They don't replace other design methods.

Individual patterns provide the building blocks, like self-contained objects or subroutines. Some of these building blocks are so big that they structure the entire solution. Within many patterns, you'll see a section of implementation guidance, which lays out the steps you should follow to implement that pattern. The software design methods in Chapter 3 provide only general guidance for designing your system.

Sometimes, the implementation steps in a pattern mention another pattern that you should use to help build up the solution. In this case, pause the implementation of the first pattern, and proceed to implement the referenced pattern (as shown in Figure 8-3). Sometimes the first pattern will give you guidance for when you should implement the referenced pattern.

Here's a pragmatic approach to creating software architecture and design with patterns:

1. **Pick a software development method.**

   This is an important choice, which guides your approach to the overall design process. Any method can work, such as the Unified method or some agile method. The goal is to have some rigor to ensure that you don't forget things or make obvious errors.

   Patterns are supplements for a well-designed software development method, not replacements.

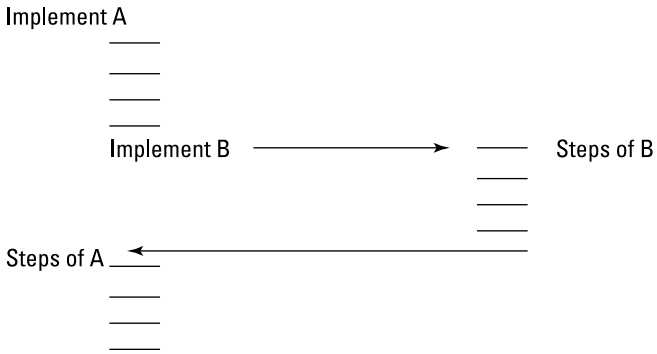


Figure 8-3: Implementing a smaller pattern while working on a larger one.

2. **Use your pattern catalog (refer to Chapter 7) to support your design.**

   As you're applying your software development method, you'll run into design problems that you need to solve. You can solve them from first principles and possibly reinvent the wheel repeatedly. An alternative — the approach I advocate in this book — is to use the proven solutions in patterns to solve problems. This method saves you effort and gives you time to solve interesting new problems.

   This pattern-based technique uses your pattern catalog from Chapter 7 and the steps earlier in this chapter, to point you to proven solutions for the problems you run into.

   Add any new patterns you find useful to your pattern handbook to keep it current.

3. **If the pattern system doesn't include a pattern for your design problem, try to find a different pattern.**

   Patterns don't yet cover all the problems you may face in software development, so in some cases you'll have to move on to Step 4.

4. **If you can't find a pattern, design a solution from scratch.**

   If no pattern exists that will resolve your problem, use the analysis and design guidelines of your software development method to create the solution that you need from scratch.

Passing through the gate

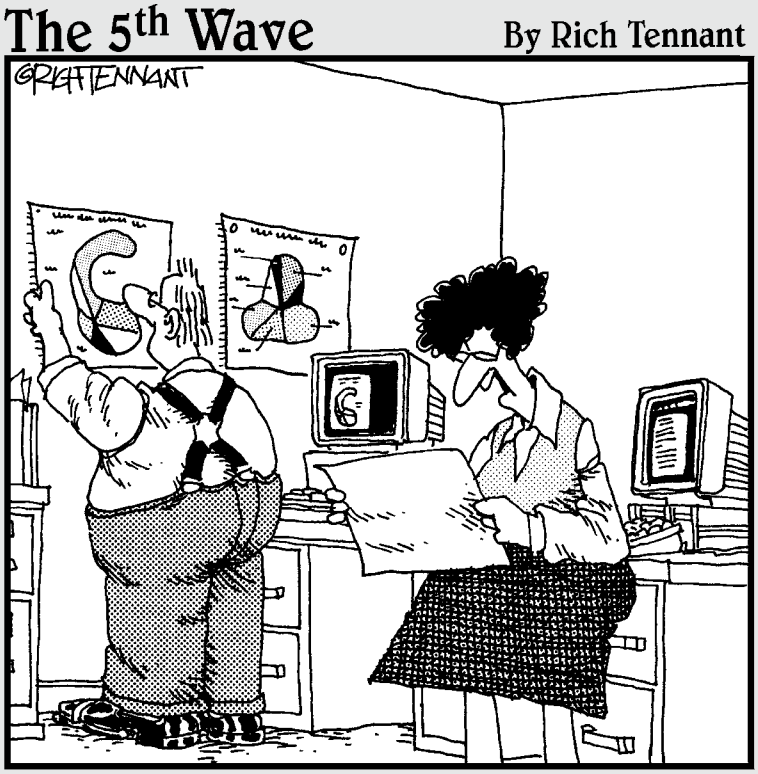
When experts talk about patterns, they talk about the notion of passing through the gate. As you work with patterns and become more familiar with them, you won't need to look up patterns to remember their implementation; the solutions they contain will become second nature. Eventually, you'll reach the point where you can't explain exactly why you employed a particular pattern, because you, too, have passed through the gate.

The gate reflects a long time of working with patterns and a thorough understanding of the domain. It isn't something that you can force your way through. You may not even realize that you've passed through the gate. What you'll find is that your peers are turning to you for guidance and instruction on those esoteric concepts that you read about in patterns once upon a time.

If you realize that you've passed through the gate, smile gently to yourself and then forget it and get about your business.

Part III

Creating Your Application Architecture



"Well, shoot! This eggplant chart is just as confusing as the butternut squash chart and the gourd chart. Can't you just make a pie chart like everyone else?"

In this part . . .

This part presents specific architecture patterns that you can use to shape the high-level capabilities of your software system. These patterns, from *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley), give you useful starting places into architectural patterns and help you solve real problems. I start by introducing patterns for structuring your architecture, followed by patterns for creating distributed and interactive systems. Finally, I present a pattern for making your systems adaptable.

Chapter 9

Building Functionality in Layers

In This Chapter

- Dividing the solution into layers
- Implementing a layered architecture

One of the most basic software architectural choices you can make is to build an architecture of layered responsibilities. Systems built with layers are all around you, from the web and Internet-enabled applications to embedded applications on custom hardware. They use layers to divide the problem into groupings that have similar responsibilities.

In this chapter, I show you how to solve a design problem by using the power of layers. To help set the stage and get you thinking about layered architectures, I start with a brief overview of three common layered architectures.

Using Layered Architecture

Layered architectures have been around since the beginning of digital computers — or at least since the early 1960s. Modern hardware technology and languages accentuate the usefulness of layered architectures, as you see in the three examples in this section. All three of these examples are in use all around you right now, in the systems you use every day.

Keeping communications open

The International Standards Organization (ISO) Open Systems Interconnection (OSI) seven-layer model (see Figure 9-1) facilitates communication between computers. The model consists of two separate but parallel stacks of layers; each layer provides a higher level of functionality than the layer below it. Within the two stacks, the layer N in one stack is a peer of the layer N in the other stack. Logically, communication is between the peer layers in the two stacks; actually, only the bottommost layer directly communicates between the two stacks.

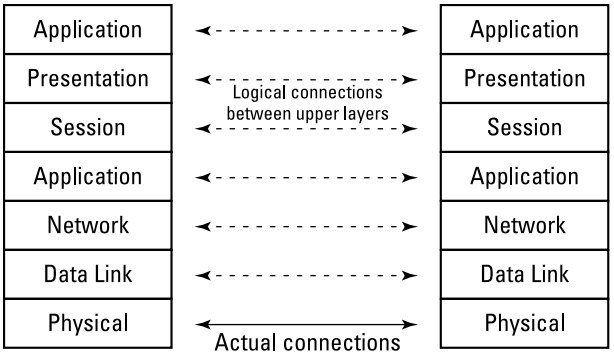


Figure 9-1: The OSI seven-layer model.

Layers can be different sizes, hosting different numbers of protocols. Over time, in fact, the communications-stack diagram has evolved into many variations. Some layers have many alternatives — such as SIP, FTP, Telnet, and HTTP in layer 7 (the Application layer) — and other layers have few alternatives.

Something that's true of layered architectures in general is true of the OSI model as well: Changes to a layer affect only that one layer. In other words, the communication protocol chosen at a lower layer doesn't affect the higher-level functionality provided at its higher level. You're free to pick and choose the protocols to use in each layer — just keep in mind that for peers to talk to each other, the peer layers in the two stacks must use the same protocol.

Creating web applications

Web applications commonly use a three-tier architecture, in which the tiers are the layers. Figure 9-2 shows a typical architecture with a presentation layer on the top, a business logic layer in the middle, and a database layer on the bottom. Each of these layers has distinct responsibilities:

- **Presentation:** The presentation layer is the web server that delivers content to the user's browser.
- **Business logic:** The business logic layer is built on something like the JBoss application server platform. It receives requests from the presentation layer, processes the requests, and supplies the results to the presentation layer. The business logic layer requests the data that it needs from the database layer.
- **Database:** The database layer manages the persistent data, such as the customer data or the online store's catalog. It supplies this data to the business logic upon request.

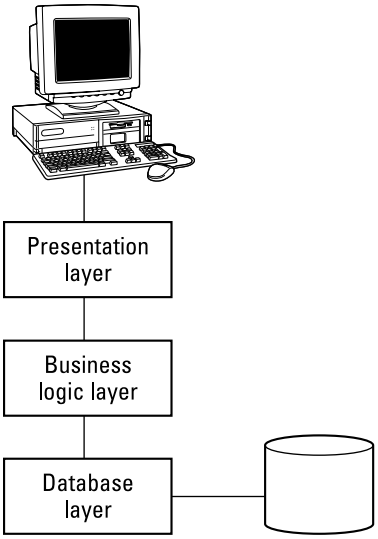


Figure 9-2: A typical three-tier architecture.

You can make changes in any of the layers without affecting the other layers as long as you preserve the interfaces among them.

Adapting to new hardware

Operating systems are examples of layered architectures (see Figure 9-3). They start with the central kernel of operating-system functionality. Surrounding the kernel are various layers of functionality, such as device drivers and middleware. The topmost layer contains the applications that the user executes.

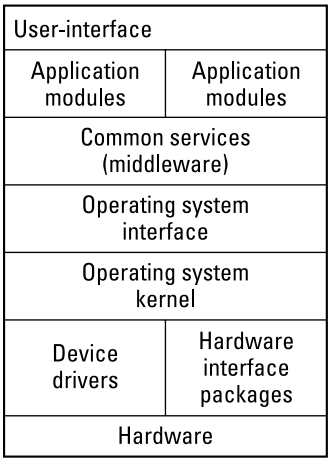


Figure 9-3: Operating-system layers.

The layered architecture makes adapting to new hardware easier, because only the device drivers need to change when a new device is added. The higher layers, like the file system, don't care about the new hardware.

Many systems replace the device-driver layer with a hardware-abstraction layer. This layer provides a common, stable interface for the kernel layer by hiding the details of the hardware and the details of hardware changes.

Problem: Designing at Differing Levels

You have a simple design problem: You're supposed to build a system that displays the state of a user's primary disk on his computer screen. This system must create a graph for the entire disk, displaying blocks on that disk in different colors to show their status: free, being used for a file, marked as bad, and so on.

This system shouldn't be hard to create, because you already know how to do the display function. You quickly identify which system calls to use, so getting the information to the display isn't hard either.

Building a monolith

You build the system as one big application, a *monolith.* The main program calls the display components to set up the display; then it iterates through the blocks of the disk, displaying each block's status. Pretty simple. In the end, you have a well-structured program that is best described as a monolith (see Figure 9-4).

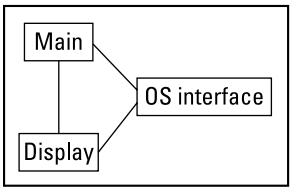


Figure 9-4: A monolithic solution.

Just when you're about to schedule the demo for the customer, new requirements arrive. Now the customer wants to display the disk mapping for a different operating system, which means that all the system calls in the main program need to be changed and need to become conditional on the operating system. The customer also wants the option to display the data as a pie chart instead of as the bar graph you provided. The changes won't be hard to make, but you didn't anticipate it.

You consider just adding the new functionality — essentially adding warts to the monolith that you've already made. These warts probably will be contagious, however, and over time, you'll have to add more and more of them as different display options arise or different operating systems need to be accommodated. The result won't be pretty (see Figure 9-5).

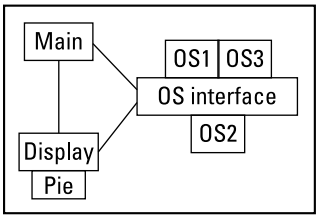


Figure 9-5: Your architecture — warts and all.

Monolithic solutions are hard to maintain, because changes aren't confined to local regions in a monolith.

Breaking up your monolith

You decide to break the monolith into parts:

- The high-level display functions sit at the top.
- The next part is the data generation part of the program, which is less abstract in that it interfaces with the operating-system functions and the display.
- The services provided by the specific operating system complete your system.

The resulting system looks like Figure 9-6.

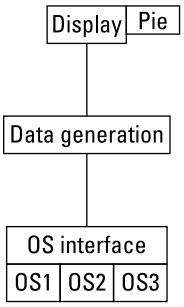


Figure 9-6: Your system, now in three parts.

Refactoring your code

Refactoring is a process in which you take the existing functionality and design of a system (or part of a system) and rearrange its internals into a new shape while retaining the same external behavior. Refactoring is done to improve some nonfunctional aspect of the code, such as making it easier to maintain or extend, usually by making a series of small changes to the code. In the disk-display example in this section, you refactor the program from a monolith — which works well in the short term — into a layered architecture that will be more flexible going forward.

Many techniques for refactoring are documented in books. Here are several common techniques:

- **Extract class:** Pull methods and data from one (or more) classes and put those methods and data in another class to refine a class's responsibilities.
- **Encapsulate field:** Make data more abstract by eliminating all direct accesses and instead requiring all access to be through getter and setter methods.
- **Pull up and pull down:** Rearrange methods and data by moving them up to a superclass or pulling them down to a subclass. This change reflects whether the rearranged methods and data are common to all the subclasses or specific to only some of them.

You've refactored your program from a monolith into a layered architecture that will accommodate change easily. (I discuss refactoring in the "Refactoring your code" sidebar.) When new operating systems are required, you can change the middle layer to adapt to new, lower-level operating-system layers while keeping the interface to the top layer constant. When new display options are required, you can add them to the top layer as new components or just revise the existing top-level component. In any case, the changes are isolated to the layer of interest. Late changes won't ripple through the entire system anymore.

Changes are confined to layers of similar responsibility.

Making this problem harder

Most, if not all, problems solved in software systems involve concepts ranging from high level to low level. The high-level operations rely on the low-level operations: The business logic accesses a database, for example, or the operating system calls a device driver. These different levels represent the groupings of the abstract concepts into the solution space. The highest level of abstraction in the solution doesn't have all the answers and can't operate in isolation; it needs to rely on capabilities provided by lower levels of abstraction.

You may think that the solution is really obvious: Just divide the software into layers. You've seen it done everywhere. A few considerations make this problem harder than it looks, however:

- Almost all the forces that lead you toward a layered architecture are related to nonfunctional requirements (see Chapter 1).
- Changes late in the development process are inevitable. You or your customer may find a better way to solve a problem, or your customer may need to add something at the last minute.
- The internal interfaces to different parts of the system should be stable, which makes it easier to maintain and extend the system. In many cases, the interfaces are going to be specified by standards, either formal standards or practices that everyone follows.
- Components aren't standard sizes; they're many sizes and shapes. Components also are complex, and not all of them are at the same level of abstraction. If you build a monolithic system, it's hard to give the complex components the attention they need without wasting effort on the simpler ones.
- Another characteristic of good design is the grouping of similar responsibilities. The most efficient way to communicate is to communicate directly between components or within a component. If you break communications into layers, you have to cross the layer boundary, which introduces communication inefficiencies. A design that has a good separation of concerns will help avoid these communication inefficiencies.

Solution: Layering Your System

Use layers to structure applications made up of groups of subtasks that are all at the same level of abstraction or that represent common groupings of responsibilities.

Exploring the effects of layers

The primary benefit of a layered architecture is enhanced maintainability. Each layer interacts only with an adjacent layer — primarily the layer below it. This arrangement means that layers may be modified, extended, and changed without creating problems for the other layers.

In the next two sections, I tell you about some other benefits and liabilities of layered-style architectures.

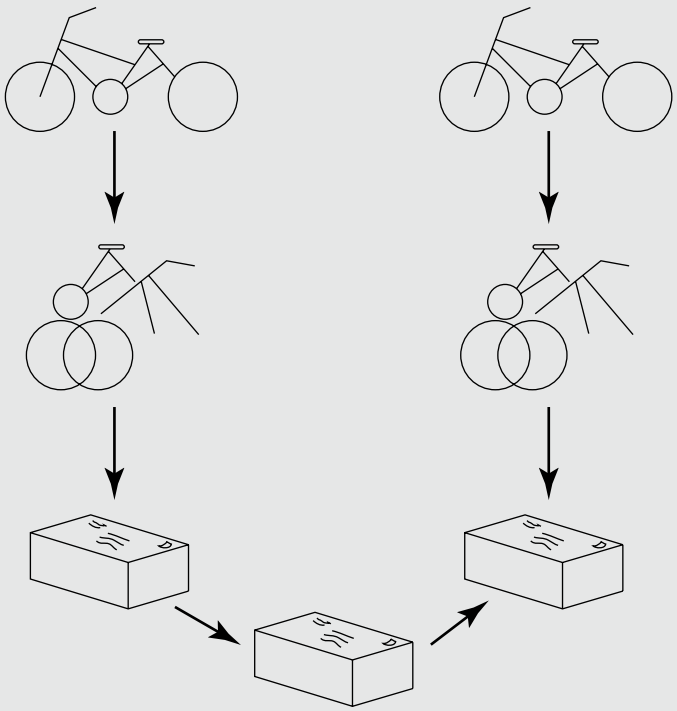
Benefits

Maintainability is the main benefit of layered architectures. There are other benefits, however, including the following:

Layering your shipments

Suppose you want to ship your bike somewhere — maybe home after a long, one-way charity bike ride. You can find people to disassemble it, pack the pieces, prepare the boxes for shipment, and move the boxes to your home. At the other end, there are people you can hire to reverse these tasks. Each of these people corresponds to a layer in your overall plan for shipping your bike.

The layers are all specialized at their task — the packer may not know how to reassemble the bike. In fact, the different layers can do what they do for things other than just bikes. Each of the people needs to know who they'll receive their work from and who they hand their output to, but otherwise they're independent and they don't know anything about your shipment.



- **Layers make reuse easier.** All the aspects that make layered architectures easy to build, understand, and maintain contribute to making them reusable:
  - Each layer has a well-defined abstraction, making it understandable and allowing you to reuse layers with confidence that they're a good fit for the situation.
  - The functionality provided by the layers is discrete and well defined.
  - The clear interfaces between the different layers make it easy for you to adapt a new problem to reuse an existing layer.

  Developers sometimes resist reusing layers because they want to write the precise components that they need. They argue that the existing layer doesn't match their needs, or they point out the performance penalties associated with communicating between layers. Reuse can be good, however, because it shortens the development cycle and allows effort to be spent on other components and problems.

- **Layers provide standardized groupings of abstraction and interfaces to a layered architecture.** Industry standards are readily adaptable to layered architectures. Standards help different groups or companies produce systems that will work together. Standards-compliant layers can be used and reused interchangeably (see the preceding item).

  A layered architecture, with its clearly defined abstractions on each layer and explicit interfaces, also can drive the definition of standards. Real-life layered systems give the standards bodies examples that they can use to show how the standard should be structured.

- **Dependencies between layers are minimized.** This structure makes it easier to isolate code changes when requirements change. In the example earlier in this chapter, after you redesign your system with layers, all the display changes are confined to one layer. This feature supports portability, because if the example system is moved to a new operating system, only the layer that interfaces with the operating system changes.

- **A layer can easily be swapped with other implementations of that layer.** Individual layer implementations that satisfy the same abstraction and the same interfaces are interchangeable without too much effort.

  If the interfaces are hard coded, you can replace the old names with the new names quite easily. If that can't be done, you can still reuse them by using an Adapter (from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley Professional]) to connect the existing layer with the new layer. If you use the Bridge pattern (also from *Design Patterns*), you can even exchange layers at runtime.

- **Layers facilitate complex development projects.** Using layers allows you to spread the work among several developers or development teams to work on the parts in parallel.

Liabilities

As with any other pattern, some liabilities are associated with using a layered architecture. You need to weigh these liabilities against the benefits to decide whether a layered architecture is right for your solution:

- **Layers aren't as efficient as hard-coded connections inside a monolithic solution.** In a monolithic solution, a component at the highest level of abstraction can call a function directly at the lowest level. This function isn't possible in a layered architecture, because all the intermediate layers are involved in the invocation. Each call handoff from layer to layer imparts a slight performance penalty to the processing. This reduction in efficiency (and overall performance) often is cited as the most significant liability of a layered architecture.
- **Protocol stack layers increase message size and add processing time.** In a communications protocol that uses the OSI seven-layer model (refer to "Keeping communications open," earlier in this chapter), each layer that handles a message adds or subtracts a new header with information for its peer layer. This activity increases message size and slows the transfer of information.
- **Changes in a layer sometimes cascade into other layers.** This situation sometimes occurs, even though layers normally prevent changes from being required in other components. An example of change cascading between layers occurs when the physical layer in a communication stack is replaced by a new physical connection that provides significantly higher performance. Moving from a 10 Mbps Ethernet layer to a 155 Mbps Asynchronous Transfer Mode (ATM) link, for example, causes changes in higher layers. The increased traffic ripples upward through the stack, with many or all of the higher layers needing changes to adapt to the increased speed at which they receive incoming packets. Some higher layers benefit from the increase in speed, which allows better-quality imaging; other higher layers have to be restructured or replicated to handle the increase in communications traffic.
- **Layers sometimes introduce unnecessary work.** This situation may occur when several higher layers request the unpacking and examination of a message. The lower layers may receive the same request several times. Another example is when several layers provide redundancy to support reliability at the next-higher levels. These multiple layers may add checksums to their messages, when one checksum would be sufficient. Because their peers must check these checksums, message sizes may explode.
- **Layered architectures have no mandatory structure for layers.** Consequently, some layered systems may have too much functionality in too few layers, which makes it harder to reuse the layers and harder to understand the key abstractions of each layer. Another drawback is that a designer can create too many layers, thereby increasing the overhead associated with the layered architecture.

Dividing a system into the most appropriate layers is a hard problem. Step 5 in the upcoming implementation section shows you how to iterate over the definitions of layers to find the most appropriate layered model for your problem.

Layering your architecture

The only component at the architecture level is a layer. Each layer communicates with the adjacent layers and is responsible for some processing of its own, passing requests to the layer below it and answering requests from the layer above it — or perhaps passing requests upward and giving answers to the layers below instead.

You can actually use layers to build five different communication scenarios or styles — three main styles and two variants. I discuss them all in the following sections.

Scenario 1a: Enable top-down communication

This scenario (shown in Figure 9-7) is the most common style of layered architecture and also the simplest. A client outside the layered system issues a request to an application on layer N. When layer N can't satisfy the request, it issues a request for a lower-level service, but it can pass the request only to layer N–1 below it. If layer N–1 can process the request, it does so; then it returns the response to the requester in layer N. If layer N–1 can't process the request, it passes the request to the next-lower layer, layer N–2, which also tries to reply. Eventually, some layer processes the request and starts a chain of replies upward. When a layer receives a reply from a lower layer that is destined for an upper layer, the reply is passed upward.

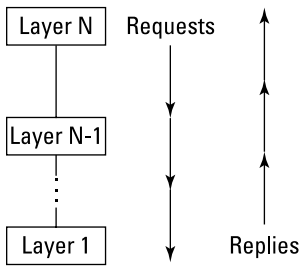


Figure 9-7: Basic layered architecture.

As a request is passed down through the layers to a point at which it can be processed, it may be split into multiple requests, each of which generates a reply. Then the replies to the divided requests are combined before being passed up to the higher layers, so as to provide a single response to the original requester.

Scenario 1b: Cache top-down communication

In all these scenarios, requests pass only from layer N to an adjacent layer. Sometimes, layer N–1 has access to all or part of the response from a lower layer and can cache that response. Later, when layer N–1 sees a request for something that it has the answer for in its cache, it can reply without passing the request downward (see Figure 9-8). This scenario provides a stateful response, the state being the cached response value. Providing this statefulness isn't easy, however, because the individual layers are harder to program.

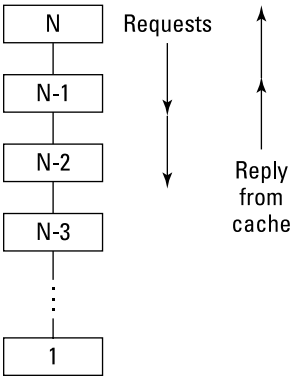


Figure 9-8: Caching results in upper layers.

Scenario 2: Use communication protocols

Another common example of layered architecture is the OSI seven-layer model (refer to "Keeping communications open," earlier in this chapter). The general structure is shown in Figure 9-9. In this scenario, two stacks of N layers communicate. Requests from each layer move only up or down in their own stacks, except in the bottom layer, which can communicate with the other stack. The layers in each stack behave as though they have direct connections to the layers in the other stack, even though in actuality, each layer passes messages down to the bottommost layer, which then conveys the message to the other stack, from where it goes up to the destination layer.

Scenario 3a: Enable bottom-up communication

In the preceding scenarios, all communications among layers move down from higher layers to lower layers. Communications also can flow upward. A device driver at layer 1, for example, may detect input that it needs to pass upward for processing (see Figure 9-10). The driver converts the input to an internal format and reports it to layer 2, which starts the process of interpreting the input. If layer 2 can't process the input, it passes the input upward to layer 3, and so on.

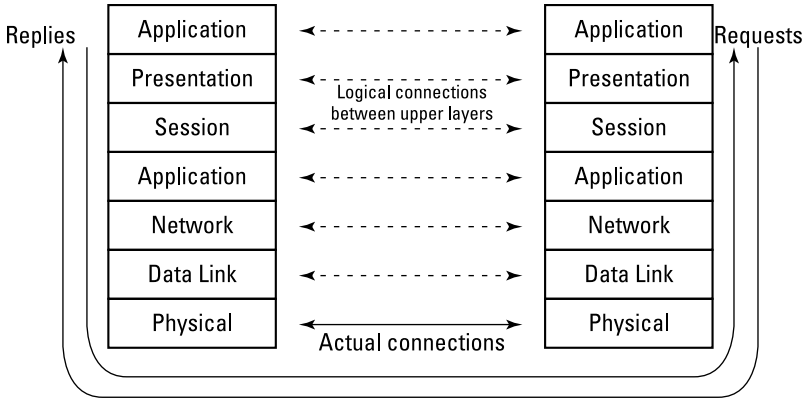


Figure 9-9: Communicating stacks of layers.

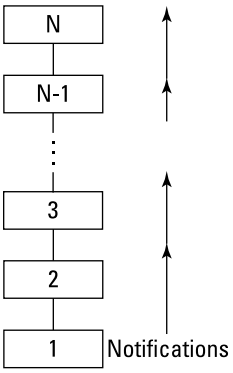


Figure 9-10: Layered notifications flowing up through the layers.

Messages passed upward are *notifications,* whereas messages passed downward are *requests.*

As in Scenario 1a, in which a request can be split into several requests as it flows downward, several notifications flowing upward can get joined as they flow upward. The lower layers always have at least the same number of notifications as the upper layers, and sometimes, they have more.

Scenario 3b: Enable stateful upward communication

In Scenario 3b, an upward-flowing notification is stopped and processed at an intermediate layer instead of traveling all the way up to layer N (see Figure 9-11). This scenario is similar to Scenario 1b, in which responses to downward-flowing requests are cached by middle layers. An example of Scenario 3b is a communications-protocol stack in which an intermediate layer determines that a duplicate

packet flowing upward should be destroyed rather than passed on, because it may confuse the higher layers. That the intermediate layer knows this information is another kind of statefulness.

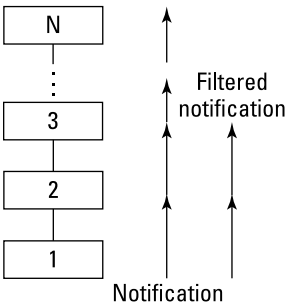


Figure 9-11: Stateful upward communication.

Implementing a layered architecture

You can take one of three approaches to designing a layered architecture:

- You can start at the top and work down.
- You can start at the bottom and work up.
- You can go up and down alternately (the yo-yo method).

In this section, I take you through a process of stepwise refinement by developing in parallel and iterating. It works in any of the three approaches I just mentioned.

You may find that you can create your layered architecture without following all these steps, and that's okay. If the software you're building depends on an external standards definition, some steps will be driven by the need to comply with those standards.

Step 1: Define the grouping criterion for your layers

In the first step, you decide how the layers generally will be organized. You have lots of ways to group functionality and components into layers, including these three:

- **Abstractions:** Abstractions of your problem can guide you in grouping the layers. Layers often are described by their distance from the hardware platform, for example. In this scenario, you have layers close to the hardware and layers at the opposite end, close to the user/client; the distance between these layers can be divided into other layers.

- **Common responsibilities:** Sometimes, layers are grouped based on common responsibilities. They contain components that are used together or used in similar ways, or that were developed by the same team or company.
- **Domain-specific functionality:** Layers sometimes encapsulate domain-specific functionalities. An example is a layer that manages interaction with a database and another layer that manages interactions with the database users.

These methods aren't independent. The domain-specific-functionality solution, for example, represents similar responsibilities and also is a way of abstracting the problem and solution into manageable pieces.

The best way to abstract a system into layers is sometimes related to the problem domain. Consider a chess-game application. In this application, the layers (from top to bottom) might be

- Strategies for the overall game
- Medium-scale tactics, such as the Saragossa Opening
- Basic moves, such as castling
- Elementary moves of the game, such as the way that the rook or knight moves

In many computing systems, the layers closest to the hardware are well defined and small because they deal with specific devices. As the layers get higher, there is more and more grouping into interfaces, services, and (finally) user-visible elements. A typical computing system can look like this (from top to bottom):

- User-visible elements and interfaces
- Application modules
- A layer of services common to many applications
- Operating-system interface layer
- The actual operating system
- Device drivers and hardware interface packages
- The actual hardware

Your system may not decompose in any of these ways. These are just common examples to get you started thinking about how to divide your system into layers.

Step 2: Decide how many layers your system will have

Each division that you find in Step 1 turns into one layer in your software architecture. As you work to decide what the software layers are, you may find that your layers of abstraction or responsibility don't precisely match your needs. Adjust!

You can repeat all the steps in this chapter until you're comfortable with the result.

Maybe the intermediate layer that provides common services should be two levels, because some of the recognizable functions rely on other functions that are closely coupled to something in another layer. Split the intermediate layer into two. Or maybe two layers interact only with each other, and you decide that you can merge them. You may even find that some nonfunctional requirements aren't met by the layering you're designing, in which case you may add a new layer to contain responsibilities related to these requirements.

Layers add complexity and delay any messaging that spans several layers, so don't add more layers than absolutely necessary.

Step 3: Name the layers, and assign tasks to them

The task assigned to the highest layer is what the user recognizes as the system: the overall system task. All the other layers help the top layer achieve the goal. As you're assigning tasks and functionalities to the layers, make sure that you include the components needed to satisfy the system's requirements. Pay attention to both the functional and nonfunctional requirements, because the system must meet all the requirements to be acceptable to the customer.

One way to determine the split of tasks in the architecture is to start at the bottom with the lowest or most basic functions. This is easier than going from the top down when you're not experienced in the best way to split the tasks. Then add infrastructure functionality to the base until you reach the highest layer.

Step 4: Specify the services

Remember that no service is split over multiple layers. Layers are strictly separated; they shouldn't access local or private attributes or functions/methods from other layers. Arguments, return codes, and error types of functions offered by a layer N should be

- Built-in types of the operating system
- Defined in layer N (or other higher layers)
- Driven by a common data definition component that is shared across the application

It's better to put more services in the higher layers than in the lower layers. This structure best consolidates the functionality and produces fewer interfaces for the developers to learn. This structure is an *inverted pyramid of reuse* (see Figure 9-12), which has a thin base layer, with each layer above the base getting slightly larger, providing more services and more functionality. If you put the same services in several multiple lower layers, the developers will have many slightly different interfaces to choose among — and then their problem becomes choosing one over the others.

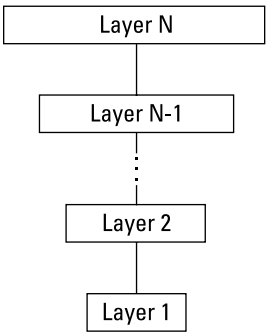


Figure 9-12: An inverted pyramid of reuse.

Step 5: Refine the layering

By now, you have a first draft of your system's layers. The layers probably can be better, however. It doesn't always work to envision the abstractions and force the layering on your vision; likewise, it doesn't always work to start designing your components and then force them into layers.

At this point, you should reflect on what you've developed so far and then revisit steps 1–4 to refine your layering.

Sometimes, alternating works well. On one pass, work through the steps from the top level down toward the bottom. On the next iteration, work in the other direction — from the bottom level up to the top.

As you refine the layering, keep in mind that layers have direct communication only with adjacent layers.

Step 6: Define each layer's interfaces

Each layer creates an interface that exposes its functionality to adjacent layers. Everything that a layer is going to be asked to do by its adjacent layers must be represented in the interface.

Black boxes

Ideally, each layer is a *black box* (hidden) to the layer above it and to the layer below it. Additionally, the layers that aren't adjacent don't know anything at all about the black box; they communicate only with the layers adjacent to themselves. The internals of the black box are hidden. For this arrangement to work, the interfaces between the layers must be clearly defined.

If a layer is already built and is being reused, it may already have a designed interface that isn't quite aligned with the needs of the new layers that you'll add to the system. In that case, you can use the Facade pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*) to encapsulate the functionality and provide the interface needed by the current system's layering.

A black-box layer provides the best reuse possibilities. Because you were forced to define the clearest, most complete interface for that layer, you can be assured that it doesn't access inappropriate services in other layers.

From a good-design perspective, it's desirable to have layers that are black boxes, but what is desirable isn't always possible. Sometimes, the adjacent layers must know something about a layer's internals — knowing that it really has multiple components that are working together to provide the layer's service, for example, or that the layer has alternative communication protocols that can provide roughly the same services.

Gray and white boxes

In addition to black boxes, you can use gray and white boxes. If a layer is a *white box,* the adjacent layer can look deeply into it, see how it provides its services, and then access those services. A *gray box* reveals some of the layer's internals but also keeps some secrets (that is, the walls of this box are gray). You can use either of these approaches as well.

Gray-box and white-box approaches are sometimes useful for improving efficiency. Instead of going through a black-box interface, the adjacent layers can access some functionality directly. The benefits of encapsulation usually are greater than any benefit from improved efficiency, but your result depends on your problem's requirements.

Step 7: Structure individual layers

The overall focus of this pattern (and of the steps so far) is to build an architecture with good, effective layering. In this step, you should examine the individual layers and design their internals. Apply all your good design practices within a layer. If the layer is complex, don't be shy about breaking it into smaller components. Nothing says that individual layers have to be monoliths (refer to "Building a monolith," earlier in this chapter). There are at least three components for each layer in the Presentation-Abstraction-Control pattern, which I introduce in Chapter 14.

Step 8: Define the communication between the layers

In this step, you look at how the layers communicate with one another. There are two basic models:

- **Push:** In the push model, a layer N pushes a request down to a lower layer N–1 for processing.
- **Pull:** The pull model works the other way: A layer N–1 asks layer N above it for the information it needs to complete its work. An example is a communications component in layer N–1 requesting from layer N the next packet of data to send.

Pulling sometimes is useful, but it can introduce dependencies between a layer and the layer above it. To prevent these dependencies, you can use callbacks, which I introduce in the next step.

Two patterns later in this book — Pipes and Filters in Chapter 10 and Publish-Subscribe in Chapter 21 — involve push and pull models.

If nonadjacent layers will communicate, document that fact so that it doesn't become a maintenance headache. Communication that bypasses intermediate layers should be limited to unusual situations.

Step 9: Decouple adjacent layers

A layer must know something about the layer below it to build functionality on top of the services in the next-lower layer. A lower layer usually doesn't know anything about the higher layers, so top-down, one-way coupling exists among the layers. That is, layer N can change without concern for its users in layer N+1 above it as long as layer N keeps its interfaces unchanged. Changes in layer N+1 don't affect layer N.

That's fine when requests flow downward (refer to Figure 9-7 earlier in this chapter). When requests flow upward from lower layers, however, bottom-up coupling can be introduced, which is harder to maintain. The system can simulate top-down, one-way coupling by using *callbacks*. Callbacks are especially effective when the lower layer is calling only a small set of functionality from a higher layer.

Callbacks work like this (see Figure 9-13):

1. During initialization, layer N+1 sends layer N a request that includes information about what interface in layer N+1 should be accessed when layer N needs to pull data from the higher layer.
2. Layer N stores the layer N+1 information in a registry that links the capabilities it needs to call in the higher layer with the callback address given to it during startup.

3. When layer N wants to pull information from above it, it sends the request to the callback address it has in its registry.

The Command pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*) explains how to turn these callbacks into first-class objects.

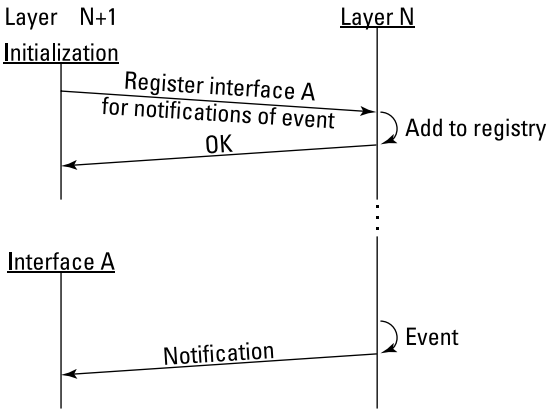


Figure 9-13: A callback between layers.

Step 10: Design the error-handling strategy

Errors need to be handled in the layer in which they occur or passed upward to some other layer that can handle them. This requirement complicates the interfaces among the layers, because in addition to performing their main functions, they need to pass around information related to errors. Another complication is that a layer N+3 may not know anything about the functionality that generated the error in layer N–3.

Here are a few rules you can follow to reduce the complexity of error handling:

- Transform errors into something meaningful to the higher layer. Instead of reporting error code 53, report that a "division by zero" occurred.
- Handle errors at the lowest possible layer to reduce the possibility that upper layers will be overwhelmed with errors.
- Sometimes internal errors shouldn't be handled at all by your software and should instead flow past the top of your layers to the runtime environment, resulting in application failure.
- Group specific errors into more-general error categories as they're being passed upward, because the higher layers will be able to handle a few more-general error conditions.

Chapter 10

Piping Your Data through Filters

In This Chapter

- Streaming your data through a series of filters
- Weighing the benefits and liabilities of a Pipes and Filters architecture
- Creating and using Pipes and Filters

In this chapter, I tell you about the pattern Pipes and Filters, which defines an architectural style for applications that stream data. The style also is useful when there are small transformations that can be done to the data in sequence, such as processing data through a series of discrete processing steps for which filters already exist.

Problem: Analyzing an Image Stream

Your boss has asked you to lead the development of a new image-analysis system. The system will take input from the new image-capture system that your company is creating and produce a stream of analyzed data. As you start looking into what's needed, you realize that the image processing will take a series of stages to transform the data from the raw input into the desired output form, as shown in Figure 10-1. These transformations are filtering the data as it streams by.

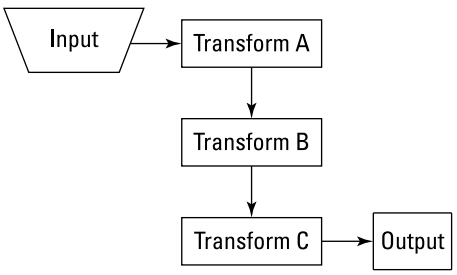


Figure 10-1: Image-analysis stages.

Your first thought is to build one big system to process all the input, as you see in Figure 10-2. The resulting system will be bigger than your workgroup can build in the required time frame. You realize that this project is an opportunity to get to know those guys in the other group down the hall better and get them to help. It also will give you more experience coordinating big projects across the organization.

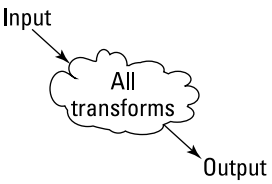


Figure 10-2: One big system that does everything.

But after thinking about it more, and after initial discussions with the other guys, you realize that the solution of one big system won't work and that you're better off working separately on each of the needed transformation stages than producing a big combined system. This realization comes after you remember that it's easier to build and test small components than huge ones.

So, instead of considering one big system that combines all the steps, you think about a set of separate transformation filters that analyze the stream one after another. This arrangement maps nicely onto the requirements view that shows the different transformations. The steps will be created as filters on the input stream. These steps filter the input by performing its required transformation and then produce output that will be used as input for the next stage, as shown in Figure 10-3.

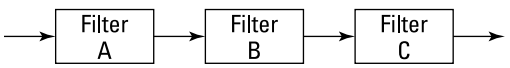


Figure 10-3: A series of filters.

This separation into filters makes development easier, too. You can focus on Filter A and Filter C, and the other guys can work on Filter B, which is their expertise. Another benefit is that you don't need to meet with them as much because your filters are independent. You know that this plan can lead to filters that don't work together, though, so you coordinate your efforts with the other team.

The filters are cohesive within themselves and have low coupling between them.

Both of your teams know how to design software to clear interfaces, like the input or message formats you agree to, so you both get started. Each team works on its own filters: Filter A and Filter C for your team and Filter B for the other team. The three parts will be able to communicate via their interface, which has been clearly specified. Carrying the data between your filters, the system uses software piping (or possibly a message queue) to create a pipeline for the data. The filters will take in what comes over the input pipe from the input source, filter it, and put the output into the output pipe, headed toward an output sink like the one shown in Figure 10-4.

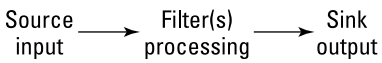


Figure 10-4: Data comes from a source, passes through filters, and is delivered to a sink.

After you've made some progress on filters, your boss points out that a new analysis step is needed. You need to perform a different transformation from the one that was originally planned, and everything needs to change. You think about the problem for a while and then realize that the solution is simple: You just replace Filter C with a new filter, Filter D. This solution proves to be an easy way to change the overall functionality.

When you initially thought about combining the parts of the system into one big system, you realized that one argument against it was the difficulty of reusing parts of that system in new contexts. The new requirements that arrived after you started development would have made for a large amount of rework.

When your boss told you that you needed to be able to solve a new and different problem, you didn't cringe, because you realized that you had the basic building blocks in the filters you'd already built. When you take the filters that have already been built and add another filter or two, you can easily solve a new set of problems. This is one of the benefits of this pipeline of components.

Even if you'd made the filters really big, you could still reuse them in different compositions. But if you'd built a single system that would combine the functionality of Filter A, Filter B, and Filter C in one big component — Component Z — and then needed only Filter A and Filter B, you'd have to start over and rebuild a system with only the two filters or build a whole new system to solve this new problem.

Small filtering steps are easier to reuse in different contexts than large steps are.

Using the pipeline of filters makes it easy to rearrange the filters. If you have Filter A that pipes into Filter B that pipes into Filter C, you can easily eliminate the Filter C step and have your solution. If you'd created one big system to do everything, the solution would be much harder and riskier. Rearranging the filters would have meant rearranging code and rebuilding the big system.

Figure 10-5 shows the overall packages of the big solution with all the transformations built into it and the packages of separate filters.

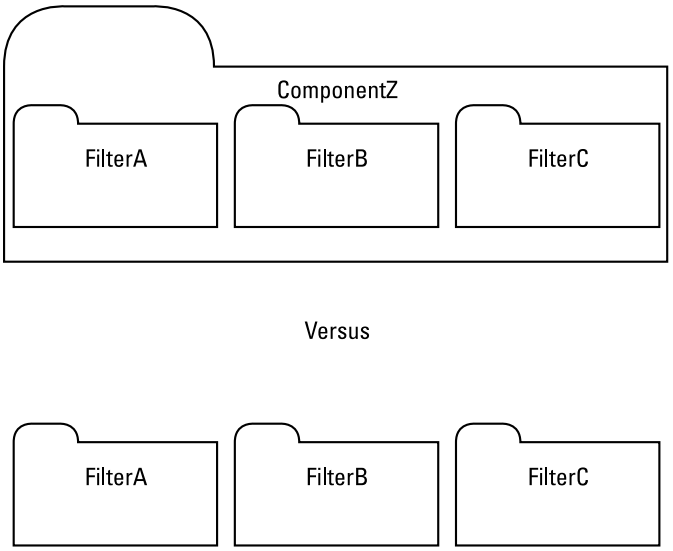


Figure 10-5: Filters can be recombined without rewriting.

After creating Filter D and adding it to your pipeline, you see that you've created a general collection of parts that can be combined in many ways.

During the process of building the system, your teams identified some benefits of your technique:

- **The amount of information that you needed to share between developers was small and well defined.** Each filter needs only a little bit of information to perform the processing it was built for, and that information is well defined and can be expressed in the interface specification.
- **There isn't much need for passing control information between the filters, because the stream of data contains all the necessary information, as shown in Figure 10-6.** I tell you more about how to handle extra control information in the "Liabilities" section, later in this chapter.

Filtering components as software tools

The filtering components in your image-streaming application allow you to do things similar to what you can do in some command shells. These components are software tools that you can combine to perform the desired functions, and you can pull these tools out of your toolbox again whenever you have a problem like the ones you've solved before.

The filters of a pipeline are perfect examples of tools. The Unix and Linux operating systems' shell programs are built on the principle of small commands that you can string together in many combinations to do new and interesting things.

Here's a little example:

```
> cat MyFile | grep MyText |
   wc -l
```

This example opens the file `MyFile` and pipes its contents to the `grep` program, which looks for the text `MyText`. Then `grep` pipes all the lines that contain `MyText` to the `wc` filter, which counts the number of lines that it receives as input and displays that number on the standard output, which in this case is the console.

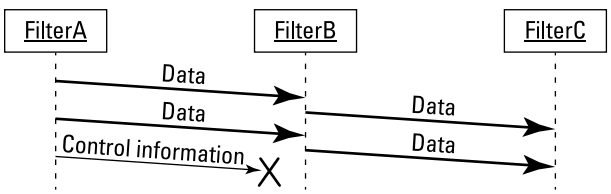


Figure 10-6: Little or no control information is embedded with the data.

Pipes and Filters work best when the input and output are clear and well defined.

Another issue was easily solved when you realized that the system can do different things to the stream of data by changing the filters in the sequence. This is what happened when your boss introduced the new D transformation. This other filter needs to know only what its input should be. It doesn't need to know what happened at previous steps in the chain as long as it gets the kind of input it expects in the format that it expects. The set of filters that you can build is quite large, as Figure 10-7 shows.

Steps in the processing chain that aren't sequential don't share information. All information flows directly from one filter to another in the pipeline, as shown in Figure 10-8. No information skips the intermediate filter elements by going out of band from one filter to another.

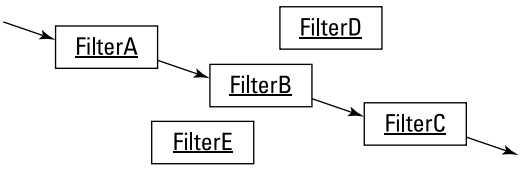


Figure 10-7: You can develop many filters for your data.

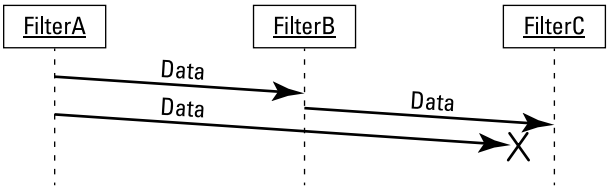


Figure 10-8: Data passes through all the filters.

Information flows only between directly connected pieces of the pipeline.

The other team's filter was designed to process input of a certain kind. It took input that was formatted in the way of the original specifications, as shown in Figure 10-9.

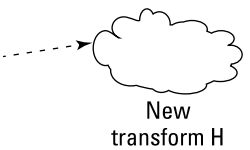


Figure 10-9: New filters work as long as they accept the same input format.

If the input should ever change — if the components were to be used for processing the telemetry from a new image data source, for example — the input format may be a little different. Figure 10-10 shows the change.

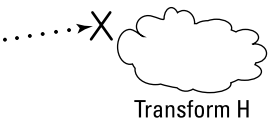


Figure 10-10: Changing the format of the pipes can break the filters.

If you add a new filter to the pipeline, however, the input can be translated from the new way into the way that will work with the existing components, as shown in Figure 10-11.

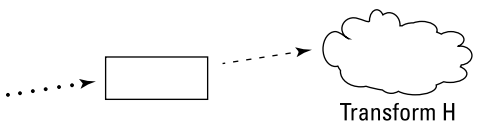


Figure 10-11: You can add a translation filter to adapt different pipelines.

The pipeline makes it easy to add new steps to acknowledge that not all data looks alike and that, even though you may want to do the same thing, such as run the same analysis, you may not be able to do so unless you allow for new building blocks. It's easy to add new filters to translate new data sources into something that your existing filters can process or to present the output differently.

You also realize that you may not want the same output all the time. You can add processing steps to the other end of the pipeline to produce different output.

During testing, your development teams needed to test your part of the system. To do this, you put sample data in a file that mimicked the output from the data source, which was easy to do. The writers of Filter B put their test input in a file to simplify their testing too. They didn't have to wait for all this new software to be done before the components were testable, which allowed all of you to develop in parallel.

About this time, your boss comes in and asks whether you're done yet. You aren't sure how to answer him. He asks why you don't deliver the filters that read from and write to files. You explain that although you could have done this, the system wouldn't have met its performance requirements. Writing a file involves the file system. Processing steps that communicate directly with one another is much more efficient because it doesn't involve disk space, disk caching, or delays. Building a pipeline to pass data from source to sink through a series of filters is a better solution. Although you didn't use message passing as the means of communications, it also would've met your performance requirements.

Explicit storage of intermediate results consumes resources.

As you're thinking about the problem space, you realize that this solution won't fit in all situations. Systems that are highly interactive (at something other than the command-line level) or that are event driven aren't good candidates because they can't easily be broken down and structured into different pipeline stages.

Solution: Piping through Filters

The Pipes and Filters pattern structures the processing of a stream of data. Each processing step is implemented as a filter, with information flowing between the filters through a pipe. Filters can be combined in many ways to provide a family of solutions.

Exploring the effects of Pipes and Filters

In any design activity, the choices that you make have consequences. This section looks at the consequences of defining a Pipes and Filters architecture.

Benefits

The Pipes and Filters architecture has the following benefits:

- **The system's behavior is very flexible.** By exchanging filters for new and different filters or rearranging the filters in a different order, you can change the system's behavior.
- **The filters can be reused in other situations.** Because the inputs and outputs are well defined and standard, you can use the filters created for one application for different applications and combine them with different filters.

Piping water to your house

The pipes that bring water to your house are a real-world example of Pipes and Filters. The water comes from its source to the first filter through a pipe. That filter chlorinates the water and directs the output through another pipe. Other filters do further purification. Some of the filters are junctions in the pipes that divert the flow or allow it to be shut off. Eventually, the network of pipes delivers the water to your kitchen sink.

The modularity of this approach allows you to easily add or change filters, maybe adding a water softener or a new sink faucet, without changing the overall system.

- **Filters and pipes help you prototype solutions rapidly because of the flexibility noted earlier.** You can easily build a prototype by combining existing filters to see whether the resulting system can be used.
- **You can use files as the pipeline, which is very convenient when you're debugging the system.** You can use files when they make sense, but you can pipe your filters directly together to achieve higher performance or greater convenience.
- **Allowing the different filter stages to talk directly through the defined interface means that you can avoid using intermediate files, which improves overall performance.** Writing information into a file and then reading it out in another stage of processing is inefficient. It takes disk storage and time to create the file, write the results, and close the file. Then the filter that's going to use the information must open the file, read it, and close it again.
- **Pipes and filters are conducive to parallel processing.** If the filters take their inputs in small portions, they can be started in parallel and run independently, allowing the work to be done in parallel. Be sure to read the next section to understand the liabilities related to parallel processing.

Liabilities

The Pipes and Filters architecture has the following liabilities:

- **It's hard to share state information in an environment of Pipes and Filters.** If state information needs to be exchanged between the different filters, it must be done *out of band* (outside the normal flow of data from one filter to another) but be kept synchronized with the flow of data in the pipe. It's awkward to use a Pipes and Filters architecture if extra information about the data being passed is needed. The alternative isn't pretty: designing an interface that combines state information and data in one stream. The state information and the data need to be combined on one side and then separated on the other side. If global data should be shared, you probably shouldn't use a Pipes and Filters architecture.
- **The gains from parallel processing of pipelines (see the "Benefits" section) can't always be realized, for several reasons:**
  - The cost of transferring the information from one filter to another may be greater than the perceived processing gain. In many systems, the actual processing still requires context switching and the management of multiple threads or processes.
  - Some filters need to consume their entire input before doing any processing, which prevents multiple filters from stepping through the data in parallel.
  - Filter synchronization may require the exchange of state or other control information, which further reduces the gains from parallelism.

- **An ideal pipeline passes data to its filters in a form that the filters can immediately process and where neither the pipeline mechanisms nor the filters need to transform the data.** This isn't always possible. Sometimes the filter mechanism, such as the command-line pipelining provided in Unix, requires that the information between the stages be translated into characters to be managed by the pipeline, which isn't always desirable or possible. Such transformations are overhead in the system that reduce its performance. To move data between filters efficiently, you can use other mechanisms such as shared memory or even direct calls handing the data off directly.
- **Error handling is difficult with Pipes and Filters because the information flow isn't conducive to reporting the error information.** If you can't find a good error-handling strategy for your application, a pattern such as Layers (see Chapter 9) may be more appropriate.

Implementing Pipes and Filters

Four different classes are present when you implement Pipes and Filters. Figure 10-12 shows these classes, along with their responsibilities.

Class Filter	*Collaborators*
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	• Pipe

Class Pipe	*Collaborators*
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	• Data source • Data sink • Filter

Class Data source	*Collaborators*
Responsiblity • Delivers input to processing pipeline.	• Pipe

Class Data sink	*Collaborators*
Responsiblity • Consumes output.	• Pipe

Figure 10-12: The classes of Pipes and Filters.

Reproduced with permission of John Wiley & Sons, Ltd.: Pattern-Oriented Software Architecture: A System of Patterns, 1996, Buschmann et al.

Implementing a Pipes and Filters architecture involves six steps. To illustrate these steps, I use the example introduced in "Problem: Analyzing an Image Stream," earlier in this chapter.

Step 1: Divide the task into a sequence of filters

These filters are the filter classes shown in Figure 10-12. Look at your problem, and identify the different tasks to be performed. The tasks should be central to solving your problem. The tasks can't have any overlap, and they must provide complete output from one filter that is complete input to another. In other words, each filter input is exactly the output of the previous filter in the pipeline. This step is where you think about other combinations or other filters that you may want to work together. Use good design principles and enabling techniques to create filters that have high internal cohesion and low coupling with other filters.

In the example, you found that the Filter A, Filter B, and Filter C components should be implemented separately.

The overall requirements of the system define the nature of the Data Source and Data Sink classes where the stream is begun and the final output is produced.

Step 2: Define the format for information that the pipe objects will pass between filters

You want the format for this information to be as uniform as possible because it allows the greatest reuse. Most filters in Unix and Linux pass streams of character that you can think of as being organized into lines. This format isn't required, though. Depending on how you connect the filters, you can use other formats, which you may want to do for efficiency reasons. It's inefficient to convert data to and from characters for the purposes of the pipe if the same internal representation is going to be used by the different filters.

Whatever format you choose, be sure to identify how the end of input will be marked so that the filters know when to stop processing.

For the information to pass between the stages of your image-stream pipeline, use a binary representation. You worked with the other team to define the data format during the development of the application programming interface (API) for each filter.

Step 3: Decide how to implement each pipe connection

The simplest way to exchange the information along the pipeline is to have each filter call the next one, pushing the data that the next filter will process toward it. If you're building on a Unix or Linux operating system, you can use the built-in pipeline semantics (|) to build up the pipeline and connect your filters. Another method is to build a framework around your pipeline that will manage the elements. This method is best if you're building a set of interchangeable filters and efficiency is a primary consideration.

There are four connection variants of Pipes and Filters, as shown in Table 10-1:

- **Push:** In the Push variant, each stage of the pipeline pushes its output to the next filter, which waits passively for its input to arrive.
- **Pull:** In the Pull variant, the final recipient pulls the data through the pipeline by requesting the information from the previous filters that keep pulling the data from the source.
- **Hybrid:** In the Hybrid variant, filters sometimes push output to the next filter and sometimes pull data from a previous filter.
- **Message-passing:** In the Message-passing variant, messages are pushed into the messaging system by one filter and pulled from the messaging system by the next filter.

Select the variant that best supports the application you're building.

Table 10-1 The Three Variants of Pipes and Filters

Variant	*Best For*
Push	Initiated by the Source having some data to be processed. Use when the amount of data is low enough that the filters can process whatever arrives.
Pull	Initiated by the Sink calling for some data. Use when the amount of data needs to be managed so that the filter asks for more data only when it's ready.
Hybrid	Filter components have varying needs within the same system. Some pull data from their source, and others push data out to the next filter. Pipelines that combine some filters that push and others that pull data are common.
Message-passing	Useful with widely distributed systems, including cloud-based systems. It's also useful for systems without Unix or Linux pipeline semantics.

In the design of your image-streaming system, use a Push variant, because the Data Source is constantly generating the data that will stream through the system. You define an interface between filter and pipe classes that packages the data from the source or previous filters and that includes necessary framing and reference information. Data will be transported in a binary representation. Any new filters must accept and produce this same data format.

Step 4: Design and implement the filters

The next step is designing and implementing the filters, which can be either active or passive.

A *passive* filter element is one that waits for its input to arrive or waits for its output to be requested. By contrast, an *active* filter fetches its own input and pushes its output out to the next stage of the pipeline. To implement an active pipeline filter, you can use either threads or processes.

When designing the filters, you should think about efficiency. The overall pipeline of information is thought of as a processing whole, but each part may be a separate process. This means that as your data passes through the pipeline, context switches will occur. The effort required to copy data between address spaces is another performance effect to consider. Creating small filters will be flexible but will increase overhead.

Also, think about how you can control and customize the filters. You may want to reuse them in slightly different ways, so consider ways to change their behavior later. Unix and Linux filters, for example, take command-line arguments. Creating a global environment that contains the control information is another method. You may create a procedural or programmatic framework to manage the Pipes and Filters, invoking them or customizing their actions. You should choose a method that is compatible with the operating system and operating environment that you're using. Because filters are built to perform only one transformation, their implementation can be streamlined and efficient.

In the example, the filters are started at the startup of the image-stream processing and given pointers to one another so that they know where they'll receive data from and where they'll provide it to. The source of the data and the sink of data will be told only where to send or receive data.

Step 5: Design a way to handle errors

Errors from within a pipeline are hard to handle in general. The individual filters have different error criteria and different rules for handling bad input or internal errors. Because there's no shared state, there's no easy way for one filter stage to report to its adjacent filters that it has an error; it should have a way to tolerate the errors and continue processing the input stream. Another thing that makes error handling hard to design and implement is the fact that the filter may have no control over the input that keeps coming in a Push variant.

Unix and Linux have `stderr`, which is a standard output stream for error information. Your filter can report its error into `stderr`. Be aware, however, that all the other filters are doing the same thing, so the record may get jumbled and hard to follow. Design the filters you're building so that when they encounter an error, they send errors to `stderr` and skip forward to the next grouping of input. To help the downstream filters realize that an error occurred, the flag indicating the error will be injected into the output stream in place of the output that would have appeared for the erroneous input. The filter elements will recognize this flag and ignore the input.

Regarding what to do with the input from an error, a good approach in a filter environment like this one is to absorb the erroneous input: Read through to the end of the line, but don't do any processing on it, and then resume with the next line of input. In some circumstances, you want an error in a filter to abort the whole processing, but that situation is rare.

The CHECKS pattern language that I mention in Chapter 23 also describes ways to handle errors in filters.

Step 6: Set up the processing pipeline

Now that the filters are built and the overall mechanism is defined, create a way to invoke your pipeline. This method can be as simple as a script to invoke a shell command line. If the system handles only a single task, you can write a program to coordinate the filters and move the data through the pipe. Using a script in your environment's scripting language is another way to define the flow of the pipeline.

For the image-processing system, you choose to create a script in which you'll specify the ordering of the filters for the image stream. This script creates the necessary references between filter and pipe objects to process its input stream from source to sink.

Chapter 11

Sharing Knowledge and Results on a Blackboard

In This Chapter

- Solving nondeterministic problems
- Letting expert knowledge sources work independently on the same problem
- Implementing a blackboard system

The Blackboard pattern has been used in the artificial intelligence (AI) community since the 1970s. This pattern is useful to give structure to the analysis of problems that don't have a deterministic solution — in other words, for solving a problem when you can't define a straight-line approach to the solution. The blackboard architecture provides a way to receive the input from multiple knowledge sources (KSes) and combine the inputs, build upon them, and synthesize all the information to achieve a goal.

In recent years, this pattern has found renewed life in game design, so this chapter discusses the Blackboard pattern as you might use it in a strategy game.

Problem: Building an Attack Computer

Your task is to create the part of a game that controls and shoots a torpedo at a target from a submarine. One goal of the game is to sink targets, and you're responsible for the attack computer (AC) portion of the overall game, which achieves that goal. (The game has other goals and other components that achieve them.) Your component directs the actions of the submarine after at least one target has been identified. Your part is *not* responsible for avoiding becoming a target or for controlling general cruising when no target is in sight.

The game reenacts a World War II–era U.S. submarine and torpedoes of that era. The torpedoes follow preset bearings until they hit their targets or run out of fuel, at which point they sink harmlessly to the bottom of the sea. When you preset a bearing in a torpedo's gyroscope, that torpedo can take a course different from the course of the submarine that shot it.

During your research on the problem, you find that the AC must ask and answer four questions to sink a target:

- Is there a ship that can become a target?
- How does the submarine need to maneuver to get into shooting position?
- Is the shooting position within the tactical limitations of the torpedo?
- Is the solution — the geometry of the target's position, the submarine's position, and the capabilities of the torpedo — sufficient to shoot a torpedo and sink the target?

The answers to these four questions are the requirements that your part of the system — the AC — needs to meet. Figure 11-1 shows the terminology and geometry that I use later in this section.

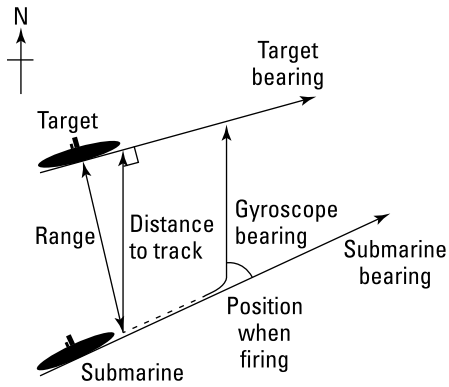


Figure 11-1: Battlefield layout.

To meet these requirements, you envision the five separate components listed in Table 11-1. There's a one-to-one mapping of questions to components, as shown in the table, except that answering the maneuvering question requires input from two components: target prediction and maneuvering control.

Table 11-1 Component-to-Question Mapping

Question	*Component of Solution*
Is there a possible target?	Target identification and selection (TI)
How does the sub get into shooting position?	Target prediction (TP)
	Maneuvering control (MC)
Can the torpedo hit the target?	Trajectory calculation (TC)
Is the solution sufficient?	Fire control (FC)

Meet the components

Each of the five components listed in Table 11-1 works part of the overall torpedo problem. I describe them in detail in the following sections.

Target identification and selection

Your AC will be activated when at least one target has been identified. There may be more than one target at any given time, so the TI needs to determine which target to attack first. After your torpedo has destroyed the target, the TI determines which of the remaining targets to attack next. If no target remains, the AC exits, and the main part of the game resumes.

The TI has only two responsibilities:

- It chooses the most appropriate target from all the potential targets.
- It performs repeated checks to make sure that the target still is a possible target.

The TI doesn't factor in the potential movements of the target, however. That job is done by the target predictor (see the next section).

Target prediction

The TP component looks at the target's movement over time and predicts where that target will be in the future. The future for the target should be short, because your torpedo will sink it, but the AC needs to know the target's short remaining life so it can plot a torpedo's trajectory to it.

Trajectory calculation

The TC component determines a course bearing for the torpedo relative to the submarine's bearing so that the torpedo will hit its target. You have to consider the following main considerations for this component:

- The TC calculates the bearing that the torpedo should take. This bearing is relative to the direction in which the submarine is pointed when the torpedo is shot and will be loaded into the torpedo before it's shot.
- Your torpedoes are effective only within a certain range, and the TC must ensure that the torpedo will be able to go far enough to hit the target.

To simplify the discussion here, I'm not going to discuss placing the sub at the right depth in the ocean to shoot torpedoes. Assume that each torpedo will be shot at the correct depth to hit the target and that the torpedo won't pass harmlessly underneath the target.

The trajectory calculation may indicate that there isn't a trajectory that the torpedo can follow to hit the target. In that case, your submarine must be moved to a different position to achieve a good straight-line path for your torpedo. The TC gives the maneuvering control (see the next section) information about where to go to get into shooting position.

Maneuvering control

The MC moves the submarine into the torpedo-shooting position required by the TC. The MC determines how the submarine must move to get into firing position and then directs that movement. Some period of cruising in a certain direction may be necessary to put the submarine into torpedo range.

Moving into position to shoot also may mean avoiding obstacles or threats. Again, to simplify the discussion, I won't talk about this avoidance behavior.

Fire control

Fire control determines the correct instant to shoot a torpedo. When the TC reports that no more movements are needed for your torpedo to hit the target at its predicted location, your system shoots a torpedo.

Ponder your approach

When you begin to structure the AC, the first thing you should think about is putting everything together in a sequential program, maybe with some loops (see Figure 11-2). Sequential processing starts with the target identification, predicts where the target will be at a certain time in the future, and then computes a trajectory to the target at that time. This trajectory may require moving the submarine into position. The AC iterates checking the target prediction with the trajectory component to see whether the endpoint of the motion has changed. Eventually, the AC shoots a torpedo at the target when your sub is in position.

Soon, however, you realize that all sorts of unexpected things can happen during this sequential approach, such as the following:

- Another, more desirable target may present itself en route to the firing position on the first target, which can result in the TI's changing the target to be attacked.
- The movement into firing position may take much longer than expected because of obstacle avoidance, which requires more maneuvering and more TP and TC calculations.
- The target may change its course, requiring that the TP phase start over.
- Another submarine or surface ship may have sunk the target, removing it from the list of possible targets.

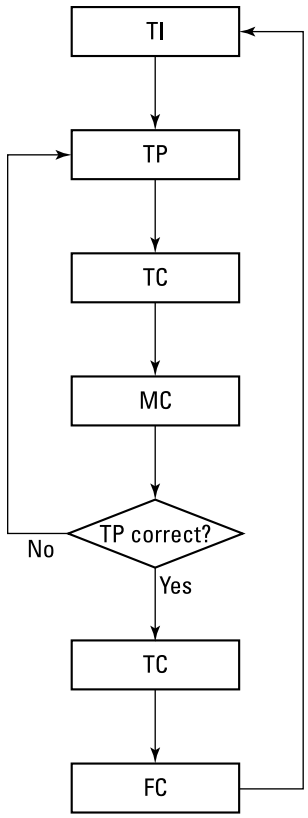


Figure 11-2: A sequential solution to the AC problem.

You realize that you'd really like all five parts of the solution — TI, TP, TC, MC, and FC — to work independently and simultaneously, and also to collaborate on the solution. You want each component to look over the shoulders of the others, as it were, and reevaluate its responsibilities based on the new information provided by the other components. The FC, for example, watches until all the other pieces fall into place. Then, when the torpedo trajectory to the target is computed for the submarine's current position and the target's predicted course, and when the target is in range, it shoots the torpedo.

Enter the blackboard

To help you think about the problem, you enlist your friends for some role playing. You ask each person to play the role of one of the five components (refer to Table 11-1, earlier in this chapter) and work on an attack scenario.

First, you try the sequential approach shown in Figure 11-2, earlier in this chapter. As you expect, the sequential approach doesn't work very well.

Next, you ask everyone to work on his or her own part of the problem, continually updating the results and making them available to everyone else by shouting. This approach degrades into chaos, of course, as everyone tries to be heard at the same time.

The room has a blackboard on the wall, so you try using it to coordinate the information. You start by putting everything you know about the problem on the blackboard (see Figure 11-3).

TI	TP	TC	MC
Target A	A: bearing 60° range 6000 yds	Torpedo range 3000 yds	Bearing 40° speed 5 kts
Target B	B: bearing 70° range 10000 yds	Torpedo speed 20 kts	
Target C	C: bearing 65° range 11000 yds		
	FC: No solution		

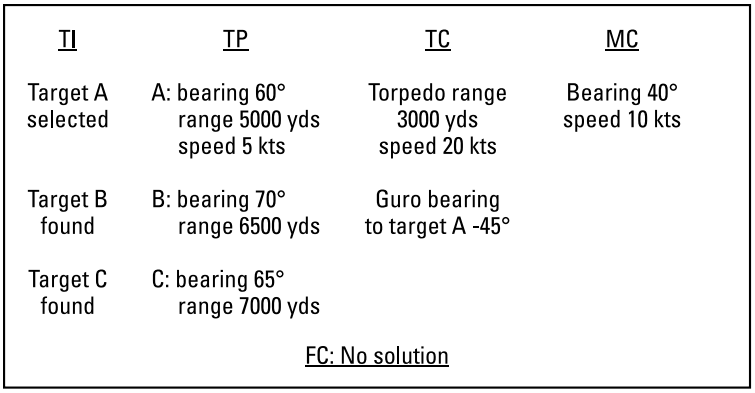
Figure 11-3: The blackboard at the start of the problem-solving exercise.

Next, you give everyone a different-colored piece of chalk and an eraser. Each person has permission to write his or her new results, predictions, or hypotheses on the blackboard, and to erase previous results and hypotheses that he or she knows aren't correct. (Because all five role players are working together and will be rewarded when the target is sunk, no one sabotages anyone else's work by making incorrect erasures.)

What the role players write on the blackboard, then, is a combination of results from algorithms that they know to be true and hypotheses that they're less certain about. Hypotheses written on the board by one role player may later be found to be incorrect as a result of some later analysis. The hypotheses are intermediate results, such as notes from the TC that the target was out of range. Even if a final result is written on the blackboard, it may be erased when some new information invalidates it.

This strategy works really well. Figure 11-4 shows the blackboard in an intermediate state.

From this example, you realize that some of the forces involved make for a hard problem: building an AC that performs its five assigned tasks. Everyone has a different way to perform his or her work and solve the problem. The MC role player, for example, never modifies or erases what the TP role player does, and the FC doesn't do much until the right time to shoot a torpedo arrives. You make these observations:



TI	TP	TC	MC
Target A selected	A: bearing 60° range 5000 yds speed 5 kts	Torpedo range 3000 yds speed 20 kts	Bearing 40° speed 10 kts
Target B found	B: bearing 70° range 6500 yds	Guro bearing to target A -45°	
Target C found	C: bearing 65° range 7000 yds		
	FC: No solution		

Figure 11-4: An intermediate snapshot of the blackboard.

- Each role player uses his or her own algorithms to do his work.
- Each role player is interested in different details of the situation — that is, different details of the data model. One player may not understand what other players are writing on the blackboard because their algorithms have different vocabularies from his.
- The players are working on results provided by the others. None of them has enough information or understanding of the problem to achieve the goal alone.
- A lot of uncertainty occurs throughout the simulation. Will the target move as predicted? Can the sub be moved into position quickly enough to achieve a desired trajectory? Would a better, more valuable, or easier target present itself while the sub is en route to the first target? All these possibilities make the situation very fluid, so you understand why the sequential approach doesn't work.
- Everyone usually is able to work on something all the time; no one is blocked waiting for another player to finish something. Constant work enables the players to reach the goal faster (although working while waiting for the movement into firing position seems to take forever).
- While the five role players are analyzing the situation and running through the scenario, you give some thought to simulating all the possible combinations of positions — both the target's and your sub's — and all the possible trajectories. You realize, however, that this approach would make the problem so big that solving it wouldn't be feasible.

TV sleuths

A staple of TV police dramas is the blackboard (or whiteboard). The blackboard is used to post all the evidence, in addition to serving as a central place where the whole team sees and analyzes the clues and leads. The lead detective controls what gets placed on the blackboard. Each person on the team is a subject matter expert responsible for analyzing some of the data. The lead detective picks the appropriate experts for each investigation depending on the situation (for example, a ballistics expert isn't needed if the murder weapon was a knife).

This approach is very flexible. The experts can add and change the leads posted on the blackboard as their personal investigation proceeds. The chosen expertise matches the situation. The experts apply their own expertise yet collaborate on solving the larger problem.

Put your blackboard into software

The next step is putting this solution to solving a submarine-attack problem into software, using what you now know about the blackboard. You know that the solution has two parts:

- **Blackboard:** The blackboard is a common shared component.
- **Knowledge sources:** The KSes will implement the decision-making processes for each of the roles.

In the simulation that you carried out with your friends, everyone was constantly thinking about his or her part and updating the blackboard. The computer doesn't work in parallel the way your friends do, however, so you add a control element — something that will keep the decision-making processes working cooperatively.

A couple of times during their simulation, your friends fought about who could write on the blackboard at any given time. The same will be true of your AC system. Even on parallel or multicore computing platforms, you have only one blackboard, and only one KS can write to it at a time. The computer system behaves as though it has only one piece of chalk, which must be passed around when KSes want to access the blackboard.

The single-shared-blackboard component has to be protected, because in a truly parallel environment, the KSes may compete for the chalk. One source may try erasing what another is just writing, thereby losing information. You must design a way for the control component to give each KS a way to say, "I want to write on the blackboard," after which each KS is given the ability to access it.

The FC KS is the ultimate authority — the only one that declares the goal (sinking the target) achievable or achieved. The FC is always watching the information on the blackboard, and it shoots a torpedo only when the MC says that the sub is at the position specified by the TC and the target is where the TP said it would be. After the torpedo is shot, the FC KS watches the torpedo's progress to see that it hits the target.

Solution: Building the Blackboard Architecture

Solve problems that have no predetermined sequential solution by using a blackboard to coordinate the intermediate results of several knowledgeable subsystems to achieve the goal through refinement of partial solutions.

Exploring the effects of the blackboard

The independence of the parts of the system and their lack of direct communication enhances the capability of blackboard architectures to satisfy nonfunctional requirements such as changeability, maintainability, and dependability. Blackboard architectures are very useful when you don't know in advance how to reach the overall solution.

Benefits

Here are the benefits of using a blackboard system:

- **The blackboard makes it easy to experiment with different ways to solve the problem.** Because there are no direct interactions among the KSes, you can change and revise them easily without affecting other parts of the system.

  There isn't any perfect number of KSes, so you can add and remove them at will.

- **A KS can be reused between solutions.** A KS doesn't have ties to any one particular blackboard system, any more than it has ties to the other KSes within the blackboard, so it can easily be reused elsewhere.

  In order to reuse KSes, they must be designed to be general — avoid tying them too closely to the blackboard.

- **Because the parts of the system are independent, a failure in one KS doesn't affect the others.** Individual failures may prevent the overall goal from being achieved, but they won't prevent ongoing work. This structure makes the solution much more robust and dependable.

Liabilities

As with all other patterns, some liabilities go along with the solution. You need to weigh the following drawbacks against the benefits to build the best solution:

- **Blackboard systems are difficult to test.** Because the system isn't following a predetermined sequence or overall algorithm, you won't always be able to determine that it's doing the right stuff to reach its goal. The information on the blackboard at any given instant may be incorrect, perhaps because one of the KSes drew an unproven hypothesis or because the KS code contains a bug.
- **Blackboards work best for problems that have no predetermined way to achieve the goal.** This benefit of the architecture is also a liability, because there's no way to guarantee that the blackboard system will generate a good solution that satisfies the goal.
- **Creating a good strategy to control the chaos of all the KSes writing to the blackboard is difficult.** In Step 6 of the implementation section later in this chapter, I present some control heuristics that you can use. As in most use of heuristics in AI, the suggestions and strategies are helpful, but they aren't guarantees.
- **Historically, real blackboard systems employed in AI have taken years to refine.** This fact should reinforce the difficulty of developing a working blackboard system that achieves the goal reliably, consistently, and quickly.
- **You still need to manage parallelism in the system, even though the KSes are independent.** The blackboard is a shared resource. Even when your computing platform can support parallel execution through multiple cores or threading, the independent elements still need to review and update the contents of a single blackboard. As a result, some part of the system — probably the control — must moderate access (or at least write access) to the blackboard. If access for reading isn't moderated, then writes must be atomic to ensure that consistency is maintained.

Knowing the parts of a blackboard system

A blackboard system has three kinds of components: the blackboard itself, the KSes, and the control. I describe them all in the following sections.

The blackboard as a knowledge repository

The blackboard is the actual repository of knowledge. It's the source of data that the KSes operate on, and it's the place where the KSes write their answers and hypotheses.

Two kinds of data can exist in a KS:

- **Static data:** Static data is information that doesn't change, such as the shoreline geography in this chapter's AC example.
- **Dynamic data:** Dynamic data, which can change, includes information such as target locations, current submarine positions, and valid torpedo trajectories.

The responsibilities of the blackboard are straightforward, keeping track of the central data for the KS on a Class-Responsibility-Collaborator (CRC) card (see Chapter 2), as shown in Figure 11-5. During the implementation steps in the next section, you see that this task is actually a complicated one.

Class Blackboard	***Collaborators***
Responsibilities • Manages central data	

Figure 11-5: Blackboard CRC card.

The information written on the blackboard — the information stored in the blackboard data store — is of various types. Some of the data is positions of targets and your submarine; some is the parameters of the torpedo trajectories; some of it is control information that's used to guide one or more of the KSes. The KSes also may use the blackboard as a scratch pad for internal information.

Not all the KSes need to be able to read all the data, but some of the information on the blackboard needs to be shared among KSes. A standard vocabulary is needed to permit this sharing. A KS must know the shared vocabulary to access information on the blackboard. Not all the information stored with the vocabulary will be relevant to all the KSes, but all the KSes must be programmed to get what they need and ignore the rest.

I mention earlier in this chapter that the blackboard may store both solutions and hypotheses. *Hypotheses* are abstractions from the current situation. Eventually, one hypothesis will be the solution or answer that the system supplies. Frequently, it's useful for the blackboard to store the level of abstraction of the information, along with the information itself. In the AC example, the raw position data is at a very low level of abstraction, and target-position predictions and submarine-movement plans are a little more detailed (see Figure 11-6). Trajectory computations are at a higher level yet, and at the highest level is the information that a torpedo is on its way to the target — in other words, that the FC has shot a torpedo.

Highest level:	Torpedo hits target
	In position to fire
	Torpedo course prediction
	Target movement prediction, submarine course prediction
	Target selected
	Target relative position
Lowest level:	Target found and identified

Figure 11-6: Levels of abstraction in the AC system.

It's useful to think about hypotheses as being *part of* or *in support of* other hypotheses. These terms help remind all the KSes of the relationships among the hypotheses. A given position data entry, for example, may be identified as being in support of a certain trajectory computation, or a target prediction may be part of a particular target selection.

Another attribute that's useful to retain with the information and hypotheses on the blackboard is its *degree of truth* — a measure of the certainty of the information. This measure helps the KSes judge the results that they can supply and comes into play in the condition part of the KSes, which I tell you about in the next section.

Knowledge sources as experts

The KSes are experts on solving certain parts of the problem. They access and use the information stored in the blackboard, generating hypotheses and conclusions based on their own algorithms. No KS can solve the problem by itself, yet each KS contributes to the system's overall ability to achieve the goal or find a solution.

KSes don't depend on any other KSes in the system to make their analyses. They communicate with one another only through the blackboard — not directly. KSes usually don't communicate with the control component, either. When a KS *does* communicate with the control element, the communication usually is about whether the KS can contribute to the solution or is about to be invoked by the control (see the next section).

Figure 11-7 shows the CRC for a KS.

KSes look both ways

Knowledge sources can operate in either of two ways:

- **Forward reasoning:** In forward reasoning, a KS uses information on the blackboard to try to make a higher-level hypothesis that is closer to the ultimate goal.
- **Backward reasoning:** In backward reasoning, a KS looks at a higher-level hypothesis and works backward through the information on the blackboard. It uses the lower-level information to reinforce the higher-level solution — that is, to increase the degree of truth of the higher-level hypothesis.

Class Knowledge source	***Collaborators*** • Blackboard
Responsibilities • Evaluates its own applicability • Computes a result • Updates the blackboard hypothesis and conclusions	

Figure 11-7: The knowledge-source CRC card.

Knowledge sources have two kinds of functionality:

- **Condition part:** The condition part of a KS is responsible for examining the information on the blackboard and determining whether the KS can contribute to the overall level of knowledge. The condition functionality must execute quickly. It doesn't actually use its algorithms to create new blackboard entries; it just does a quick assessment. Depending on the implementation, this information may be placed back on the blackboard or communicated directly to the control. The control uses the condition to decide which KS should "get the chalk" and work on the blackboard. Some designs of the control perform a quick poll of all the KSes to find their condition information before assigning one to start working; the control expects that KS to respond quickly to the condition request.
- **Action part:** When a KS gets write access to the blackboard, the KS control goes to the action part of the KS. The action part uses the information that's currently on the blackboard to create new hypotheses. Any actions, such as executing a maneuvering command or shooting a torpedo, are initiated by a KS writing on the blackboard, which serves as a record of initiation.

> The action part is allowed to execute for as long as it needs to, but it still should try to do its work quickly. Depending on the computing platform and the characteristics of the information acquisition and control strategies, the action part may pause mid-execution to allow the control to invoke some other KS that has a condition with much higher priority than that of the pausing KS.

Blackboard system control

The control component of the blackboard system runs in a loop, continually evaluating the current state of the blackboard and invoking the KS that brings the most benefit to the overall solution. The control is such an important part of a blackboard system that getting it right is a difficult task — one reason why blackboard systems are hard to implement. In implementation Step 6, later in this chapter, I explain strategies that the control can use to decide which KS should get the chalk.

The control uses information stored on the blackboard. The control sometimes receives direct input from a KS — typically, from the condition part. The control hands the chalk to a KS and gives it the ability to change the blackboard.

Figure 11-8 shows the CRC card for the control, and Figure 11-9 shows the overall UML class diagram for the blackboard architecture.

Class Control	***Collaborators*** • Blackboard • Knowledge Source
Responsibilities • Monitors blackboard • Schedules Knowledge Source activations	

Figure 11-8: The control-component CRC card.

Sometimes you want to include a special control KS in the system in addition to the control component that isn't in a KS. This KS performs an overall evaluation of the current state of the system to guide the control in making its choice of which KS to run. The non-KS control component can use the condition-part information from the KS, or it can look strictly at the state of the blackboard, without special input from the KS.

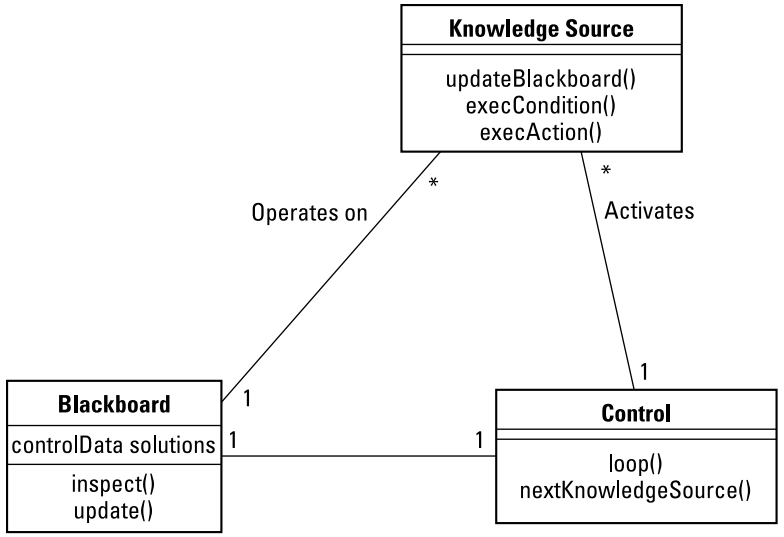


Figure 11-9: The blackboard structure class diagram.

At any given instant in the execution of the system, at least one KS should have something to contribute — adding a new hypothesis, changing another, or reporting a solution. If the system reaches a state in which no KS has anything to change, the control should signal that the system can't produce a result that satisfies the goal. More commonly, the number of possible hypotheses and KSes wanting to add to the solution grows larger — a situation that requires the control to pick the KS that's best able to contribute to the solution.

The control usually relies on a special KS to indicate that the goal has been achieved. In the AC example, this KS is the TI KS (refer to Table 11-1, earlier in this chapter), which decides that there are no longer any targets.

Implementing a blackboard architecture

In this section, I show you how to implement a blackboard architecture, using the AC system as an example. Getting this architecture to work well is a complex process. Both your engineering judgment and your intuition are required to balance the forces and create a working system. I highlight some traps and pitfalls as I go along.

Step 1: Understand the problem

Blackboard systems are most useful when information is ambiguous, the path to the solution is not known in advance, and the solution is not deterministic.

To overcome the difficulties inherent in these kinds of problems, you need to thoroughly understand the problem space, perhaps by following these steps:

1. **Determine the general fields of knowledge needed to find a solution to the problem.**

   You want to have KSes for all these general fields. In the submarine example, you have these general fields:

   - The geometry of the objects relative to one another, their directions of motion, and their speeds
   - Motion prediction to understand how the target, torpedo, and submarine will move
   - Target identification
   - Route planning to get from one point to another

2. **Understand the inputs to the system.**

   Your decisions while implementing the KSes and control will be guided by an understanding of how the inputs vary. Do they come together quickly, as when the target is fast moving, or are they slow? Are there any external inputs, or is the problem totally one of understanding the initial state of the blackboard (as may be the case in a speech-recognition system)?

3. **Define the outputs of the system.**

   Decide what indicates a successful result and how you'll know whether a result is incorrect.

4. **Determine how the users will interact with the system.**

   Sometimes, a human user can guide the system by providing inputs. Essentially, you use the human as a KS.

Step 2: Define the solution space for the problem

In this step, you define the levels of abstraction in the solution space. You must decide what a top-level, ultimate solution or goal would look like. Torpedoes shot and targets disappearing (destroyed) are the top-level abstractions for your AC.

Use the information you have about the raw input information from Step 1 as the lowest level of abstraction of your problem. What intermediate solution levels are there? In the AC problem, you have intermediate levels of courses plotted and predicted, as well as torpedo trajectories computed. Refer to Figure 11-6 to see the levels of abstraction for the AC problem.

Within the solution space of your problem domain, you may have both complete and partial solutions — that is, you may not have a one-to-one mapping between solutions and levels of abstraction.

Step 3: Divide the solution process into steps

In this step, you need to understand how the information coming into the system is transformed into a solution to the high-level problem. Each of your KSes creates hypotheses based on the information that's available on the blackboard. These hypotheses are partial solutions at the problem's different levels of abstraction. Each KS stores its hypotheses on the blackboard for the other KSes to see.

Each KS verifies that the hypotheses that it has made are correct, based on other updated data on the blackboard. It also looks for useful information and hypotheses placed on the blackboard by the other KSes. Each KS uses its own knowledge and world view to synthesize the information on the blackboard and then propose, refute, or support hypotheses.

The system succeeds when one of the KSes finds the overall problem hypothesis is true, which identifies that the solution has been reached.

During this step, try to identify any kinds of knowledge that would stop different lines of reasoning and indicate that no solution exists. These heuristics can speed the process by eliminating dead ends.

Heuristics are tricks that have been acquired over time and through applications of similar systems. It's common-sense wisdom that blackboard system designers have programmed into controls.

Step 4: Sketch the knowledge sources

In this step, decide what the subtasks and knowledge sources must do. Determine each KS's basic algorithms and how those algorithms will work — in other words, outline the responsibilities of the KS. You need to do this before you can define the vocabulary that the KS will use to exchange information (see the next section). You actually design the KSes in Step 7, later in this chapter.

Step 5: Define the vocabulary of the blackboard

This step defines a way of expressing the solution and the data that goes into the solution in such a way that all the KSes can access, review, and process all the information available on the blackboard.

Individual KSes don't need to understand all the information on the blackboard, but they have to have enough knowledge to recognize valid data — and to understand what data doesn't apply to them. In some cases, you can create translators between blackboard vocabulary and internal representations used within a KS.

The control component must be able to understand everything that gets written on the blackboard so that it can make effective decisions about where to "pass the chalk" and know whether a solution has been achieved. In addition, the problem-related vocabulary must include information related to control flow and degree of truth for each hypothesis (refer to "The blackboard as a knowledge repository," earlier in this chapter).

Development of the vocabulary may iterate in parallel with the steps for defining the control and KS components. Vocabulary definition must stabilize, however, before you can complete the design of those other components.

Step 6: Design the control

The control component keeps the whole blackboard system on track. In this step, you design that component. In general, the control follows a model of opportunistic problem solving, giving the parts of the system that have the highest probability of reaching the solution at any moment the ability to work.

The control reacts to changes made by the KSes on the blackboard. Using the information on the blackboard, it decides how to pick the next KS to execute.

The control uses heuristics in its decision making to shortcut dead ends, give hints as to which KSes are most likely to have a solution soonest, and identify the hypotheses that are most likely to be useful in solving the problem.

Following are some example heuristics that you can use (but keep in mind that not all of them are useful in all situations):

- **KS priority conditions:** The KS condition-part responses to the state of the blackboard are used to decide which KS to execute next. If a KS indicates that it's on the verge of the final solution, it should go first. If you implement your KS ready system as a queue, the priority should be stored in the queue. As KSes return their condition-part responses, those responses are put in the queue based on priority, rather than order of receipt. To apply this heuristic, if the MC reports that it's in position, the FC should be executed next — enforcing a priority that the MC is done before the FC.
- **Hypothesis preference:** Sometimes it's effective to focus on hypotheses that are more likely to change by giving the KS that monitor them a higher priority. The MC's hypotheses about the sub's position sometimes would be given preference over the TP's prediction of the target position, for example. As you get more experience with the game, you realize that one or the other is appropriate — and you put it into the control as a heuristic.
- **Hypothesis scope:** In some situations, it's useful to focus on hypotheses that address large parts of the problem. This heuristic focuses on the TC's role because it builds on the hypotheses from both the TP and MC.

- **Island driving:** In this heuristic, some hypotheses are thought to be correct. Preference in execution is given to the KS that will build on these hypotheses. This heuristic is called *island driving* because processing travels from one island of certainty to another. Because the course of the target is unpredictable, and because the TP is such an important part of the AC problem, an island-driving heuristic that favors the submarine's motion isn't a good choice because it may miss crucial target changes, and one that favors the TP may miss actions on the part of the MC.

Step 7: Implement the knowledge sources

In this step, you build the KSes. Each KS has the two parts that I mention in "Knowledge sources as experts," earlier in this chapter: condition and action. The condition part evaluates the information on the blackboard, looking for information that can help it solve its part of the problem. The action part is the forward-moving part that offers new hypotheses or changes hypotheses that are already on the blackboard.

The KSes in a system don't all need to use the same technology. A particular blackboard system can have KSes that are implemented the same way with objects, neural nets, procedural techniques, and so on. Every KS can be unique in the way that it's made with respect to its peers. You can use the other architecture patterns, such as Layers (see Chapter 9) or Reflection (see Chapter 16), for this purpose.

Consider the attack computer's TP KS as an example. It bases its decisions about the target's predicted location on the observed positions of the target. It saves one previous position (bearing and range) of the target on the blackboard, as well as the target's computed course and speed. The condition part performs a quick check of the target current position to see if it's consistent with the predicted location based on the blackboard's information. If it doesn't match, then the TP signals that it should execute soon. The action part predicts the current course and speed of the target given the data on the blackboard and stores the information on the blackboard.

If your overall system is object-oriented, you can wrap non-OO KSes by using the Facade pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley Professional]).

Using variants of Blackboard

A *repository* is a variation on the Blackboard pattern. By *repository,* I mean a system that's like a blackboard and that can be implemented as a database. What's different about the repository architecture is that it has no internal control component. All decisions about which KS to execute and which hypothesis to select come from outside — either user input or another program. In the AC example, the blackboard could be a repository of the information; the player of the game could evaluate all the data and then decide what use to make of the information and which KS to invoke.

Other examples of repository systems include programming environments that combine many tools but let the user decide the course of action. Modern compilers also are repository systems because they collect shared information such as symbol tables and abstract syntax trees.

Chapter 12

Coordinating Communication through a Broker

In This Chapter

- Adding a middleman to remove dependencies
- Seeing how a broker system operates
- Implementing a broker system

In this chapter, I tell you about the Broker pattern. The concept is easy to understand: It's a component that locates the right service for a request.

Problem: Making Servers Cooperate

For this problem, you've been asked to build a system — actually, a collection of component subsystems that work together — that provides your company's services to its customers. It won't be a unified system in which all the components know about one another and know which ones to call for their services. In this case, the components that provide service are supported on multiple operating systems and hardware platforms. All the components are independent, yet they cooperate to provide the overall service. The components can act as clients within the architecture because they need services performed by other components, too.

The system has been in service for a while and now is in version 2, but providing the necessary interservice communication among the components has become overwhelming. Adding and replacing components is hard because those components need to be registered with all the other components with which they communicate, either to request services or to provide services. Consequently, the number of communication paths in the system has exploded (see Figure 12-1).

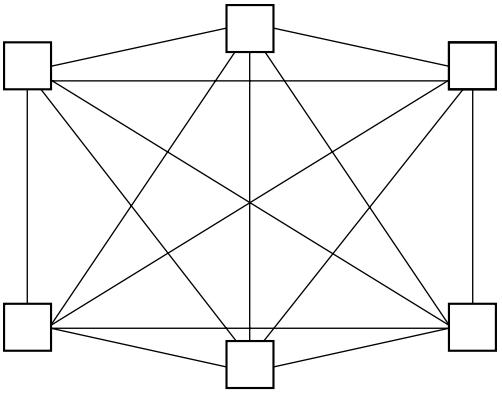


Figure 12-1: An explosion of direct-connection complexity.

In version 1 of the system, all the components were collocated in one large program — a program that had all the problems that go along with a large monolithic solution (see Chapter 9). The program was inflexible and hard to maintain. Although the version 2 distributed solution solved some of the most difficult problems in the original centralized system, it introduced new problems of its own.

Now you're going to redesign the overall system, creating version 3. The goal of this new version is to facilitate maintenance, evolution, and change — even at runtime. You plan to divide the system into parts that can be spread around your development teams efficiently so that the teams can concentrate on the parts they know best. The developers won't know what other components will use the components that they're creating. The style of application programming interfaces (APIs) and the actual interfaces will change from version 2, but the overall internal structure of their components will remain the same.

Thinking about the problem

You know that you don't want the component developers to have to worry about the overall solution architecture. After making that decision, you start thinking about the other requirements:

- **The components need to be able to find and access the services provided by other components quickly and easily.** The components shouldn't have to worry about where these other components are located — locally or remotely. Locations should be transparent to all components.

- **The system should allow runtime changes in the components being used.** This feature allows you to upgrade the components while the system is running, either to provide new functionality or to fix faults in the components.
- **Users of components — either your company's customers or internal components — should be shielded from the details of the other components.**

One of the first things you think about is using the Mediator pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley Professional]), shown in Figure 12-2. A pattern like this one may be useful, but Mediator was designed for use in small systems and for mediation between classes. You realize that the problem you're solving is a system of *systems,* rather than a system of classes.

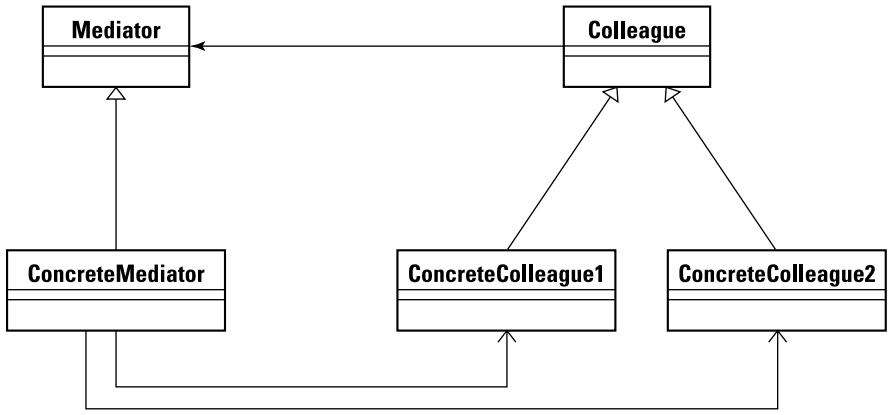


Figure 12-2: Classes involved in building the Mediator pattern.

Adding a middleman

Although the Mediator pattern isn't right for this solution, you see the benefits of a centralized middleman component (see Figure 12-3). The middleman receives requests from clients, finds the component that can provide the requested services, and then passes messages and responses back to the client. When you sketch out the message flow (see Figure 12-4), you see that this setup is similar to the Mediator pattern, but as I note in the preceding section, the parts are components rather than classes.

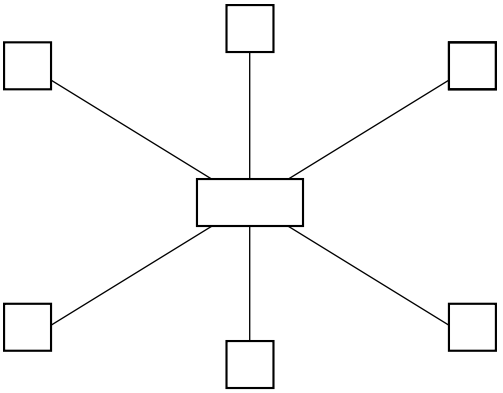


Figure 12-3: Streamlined communications without the complexity.

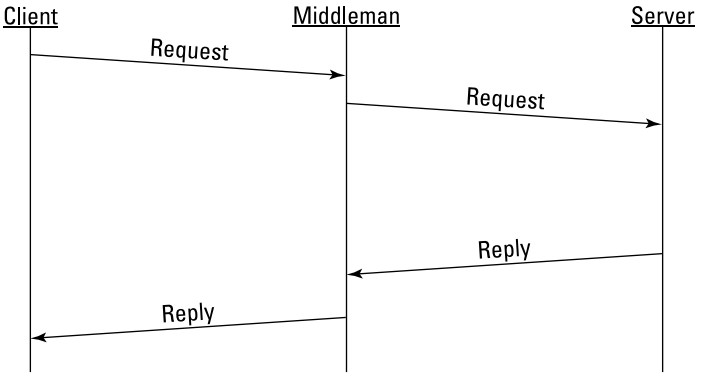


Figure 12-4: A UML interaction diagram of a middleman.

You start sketching the structure of the solution. In the center is the middleman, or *broker.* The broker is responsible for

- Locating servers to provide services in response to requests from clients
- Registering and unregistering servers, and keeping track of which ones are available for service
- Conducting messages from client to server and back
- Managing error recovery
- Connecting different systems by communicating with other brokers

Connecting clients and servers

The different clients and servers don't know where the other servers are located, but they do know that some server in the system can provide the services that they need. They don't have the necessary information to communicate directly with the other servers; instead, they pass messages to the broker, which forwards the message to the destination.

Knowing where the services exist is one of the big problems that the broker must deal with — a problem that's solved by registering the servers with the broker. Figure 12-5 shows a typical message exchange for registering a server with the broker.

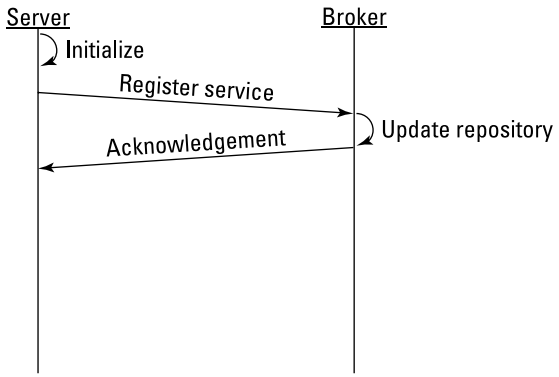


Figure 12-5: A server-registration scenario.

After the initial registration, communication between the clients and servers goes through the broker, which is now the central communication hub. The broker knows what servers can help each client, and it keeps track of all requests in the system to route them correctly to the requesting clients (see Figure 12-6).

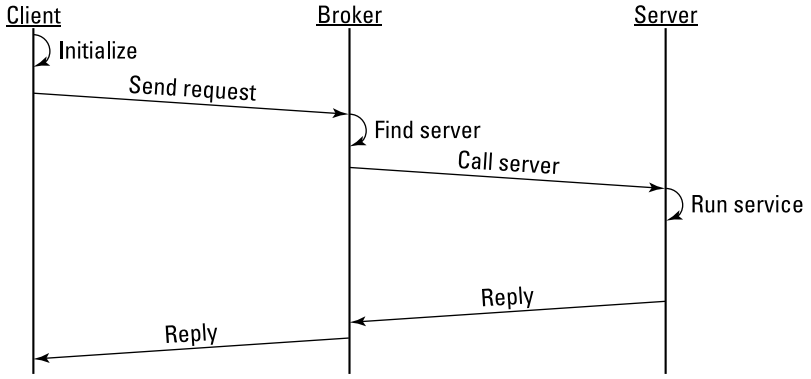
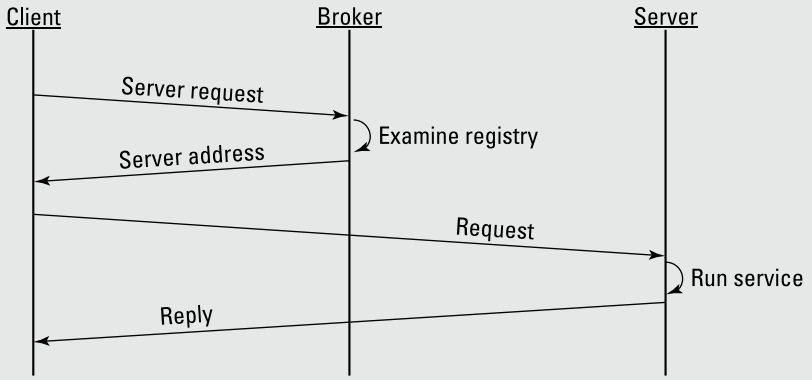


Figure 12-6: A request-and-response message scenario.

Going for broker variations

You can choose among several variations on the standard broker system:

- **Direct communication:** The direct-communication broker system supports a direct communication path between client and server. The broker still plays a role in connecting client and server, telling them how to reach each other. After making the connection, the broker steps out of the picture and lets the client and server talk directly. This kind of messaging sequence has efficiency advantages, because the broker isn't involved in every exchange. The following figure shows a typical request flow for this variation.



- **Trader:** In the trader broker system, the client's request isn't sent to a particular *server;* it's sent to a specific *service.* The broker keeps track of what servers can provide the desired service. In this variant, the client doesn't keep track of a server identifier; it keeps a service identifier.
- **Adapter:** In the adapter broker system, the broker's interface toward the server is hidden by an additional layer. This adapter layer is controlled by the broker. (You can use more than one adapter, if you like; using multiple adapters increases the flexibility of the system.) If some of the clients and servers are collocated on the same system as the broker, the adapter arranges for direct code linkage between the server and clients, which allows very fast communication. If other servers are on other hosts, the adapter layer provides the needed interprocessor messages that the communication will need. If there were no adapter, the same communications mechanism would be used in both of these cases, which may result in much slower performance than is theoretically possible.
- **Callback:** A reactive system can use a callback broker system — a variation that works well when the system is event driven and reacts to events. The broker makes no distinction between clients and servers in this model. When an event arrives, the broker invokes the callback method of any component that has registered to be notified of the event.
- **Message passing:** If the information flowing from client to server is primarily data rather than service requests, you can use a message-passing broker system. In this system, each message includes an identifier, which the server uses to determine what action it should take with the data.

The message contains both a sequence of raw data and additional information describing the message type, message structure, and other relevant attributes that help the server understand the request.

You can combine these variants — use the direct-communication variant in conjunction with a trader system, for example. Instead of linking the client and *server* directly, this combination links the client and *service* directly.

Solution: Use a Broker

Structure distributed systems so that the components communicate via remote service invocation. A broker component coordinates communication of requests from client to server and also coordinates returning the results from server to client.

Looking inside a broker system

Three main components are involved in a broker system: the broker, the server, and the client. Figure 12-7 shows a simple class diagram.

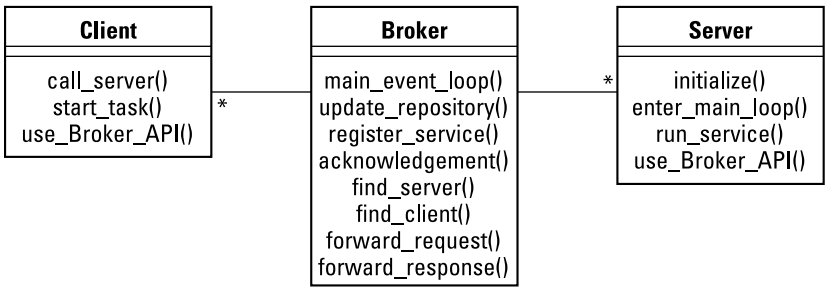


Figure 12-7: A broker system at its simplest.

In addition to these components, a broker system features proxies and bridges. I describe all these elements in the following sections.

The broker

The *broker* is the message-routing component of your system. It passes messages from client to server and from server to client. These messages are requests for services and replies to those requests, as well as messages about exceptions that have occurred. The requests are coded as calls to the broker's API. The broker is responsible for error handling in response to these exception reports.

Figure 12-8 shows the Class-Responsibility-Collaborator (CRC) card for the broker. (I introduce CRC cards in Chapter 2.)

Class Broker	***Collaborators***
Responsibilities • Register and unregister servers • Provide APIs • Transfer messages • Error recovery • Interoperate with other broker systems through bridges • Locate servers	• Client • Server • Client-side proxy • Server-side proxy • Bridge

Figure 12-8: The broker CRC card.

The broker must be able to locate the servers to which it sends requests, so it maintains a registry of the servers and their locations. The operations for registering and deregistering server locations are provided in the broker's API. If the server is local to the broker, the broker usually is also responsible for starting the service when its first request arrives. If the server is primarily connected to a different broker, the broker uses the bridge component to pass the request to the second broker for processing. (I discuss bridges in "Proxies and bridges," later in this chapter.)

The server

The *server* provides services to the clients. There are two kinds of servers:

- **Servers that offer the same commonly used services to multiple environments:** Instead of implementing a specific service many times, the system implements that service once and then shares it.
- **Servers that offer a single specific functionality to a single environment:** These servers are designed, built, and refined to perform one particular function very well, so the time and effort you invest in creating a useful reusable server are time and effort well spent.

The interfaces to these services are defined with an API or an application binary interface (ABI). In the implementation section ("Step 2: Decide the level of interoperability"), later in this chapter, I give you the details you need to decide which type to use.

Figure 12-9 shows the server and client CRC cards.

Class Client	Collaborators • Client-side proxy • Broker
Responsibilities • Implement user functionality • Send request to servers through client-side proxies	

Class Server	Collaborators • Server-side proxy • Broker
Responsibilities • Implement services • Register itself with local broker • Send responses and exceptions to client through server-side proxy	

Figure 12-9: The server and client CRC cards.

The client

The *client* is an application component that needs the service of at least one server. When it needs a service, the client puts its request into a message and sends it to the broker, which routes it to the appropriate server. Then the client can do either of two things: suspend processing and wait for the reply from the server, or continue processing and process the reply from the server when it arrives.

Clients need to know what server or service they want; they don't need to know where it's located. The broker handles locating the server.

The roles of server and client are dynamic, and components can play either role at any time. Servers can be clients and request action from other servers.

Proxies and bridges

Two other components make the broker architecture more flexible and more maintainable: proxies and bridges. To see how these components fit into a broker system, compare Figure 12-10 (which shows the class diagram for a broker system with proxy and bridge classes) with Figure 12-7, earlier in this chapter (which shows a simple broker configuration).

Proxies

Proxies hide implementation details from the clients (or the server). Sometimes proxy components are needed between the client and the broker or between the broker and the server. Proxies hide the following:

- The interprocess communication mechanism in use
- Memory-related information
- Details on marshaling parameters and results (which I explain later in this section)

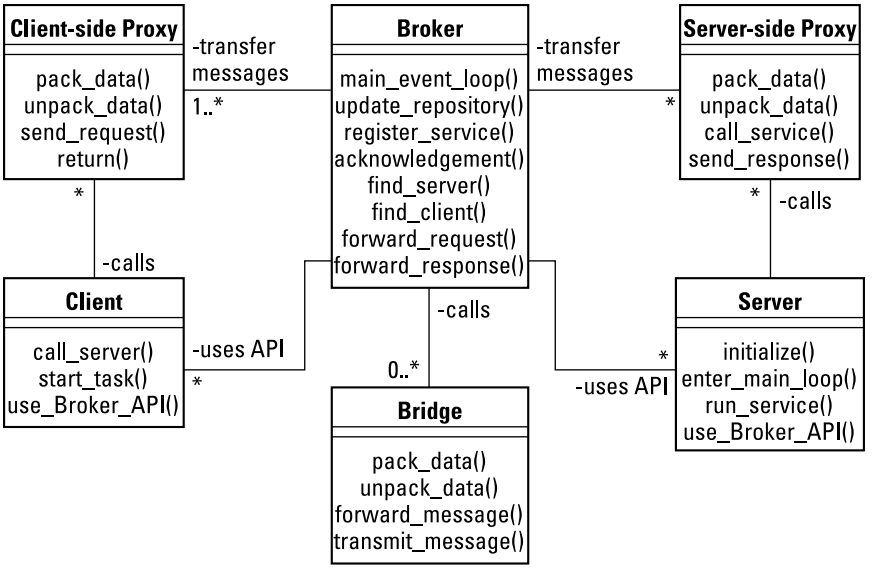


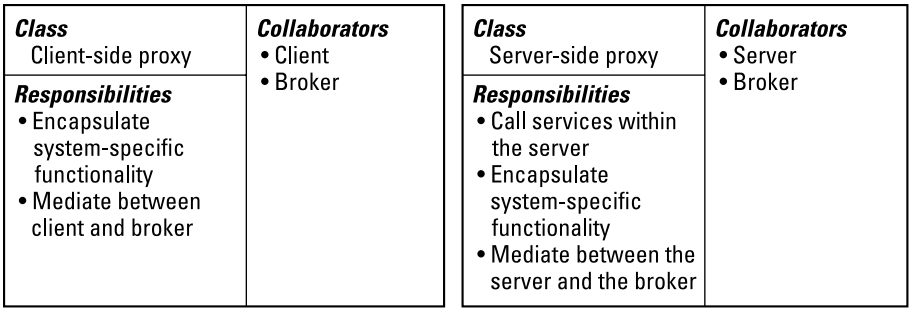
Figure 12-10: A class diagram of a full broker system.

Proxies have three main functions:

- Handling communication between clients or servers and the broker
- Translating the object model of the client (or server) into the object model expected by the broker architecture
- Marshaling parameters to go into requests and unmarshaling data from replies

Today's programming languages give you great flexibility in defining internal data structures. Communication channels and storage devices generally require streams of serial data — one bit after another. The process of going from internal representation to a serialized representation is called *marshaling.* At the other end of the process, the serial stream is *unmarshaled* into the shape of the original data structures. Marshaling and unmarshaling are responsibilities of the proxy, which serializes the messages to prepare them for the communication channel and sends them to their destinations.

Proxies can be on either side of the broker: client or server. The server-side proxy has an additional responsibility to call services, as you see by comparing the client- and server-side proxy CRC cards shown in Figure 12-11.



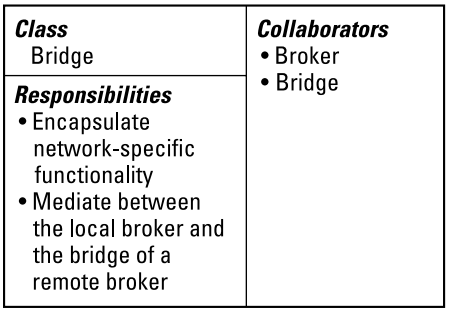
Class Client-side proxy	Collaborators • Client • Broker
Responsibilities • Encapsulate system-specific functionality • Mediate between client and broker	

Class Server-side proxy	Collaborators • Server • Broker
Responsibilities • Call services within the server • Encapsulate system-specific functionality • Mediate between the server and the broker	

Figure 12-11: Proxy CRC cards.

Bridges

Sometimes when you implement a broker system, you find that a server you need for some part of the execution is in a different broker system. To connect different broker systems, you use *bridge* elements. Figure 12-12 shows a bridge CRC card.



Class Bridge	Collaborators • Broker • Bridge
Responsibilities • Encapsulate network-specific functionality • Mediate between the local broker and the bridge of a remote broker	

Figure 12-12: A bridge CRC card.

The bridge encapsulates the network-specific functionality and mediates between the local broker and the remote broker, which may have different system-specific characteristics. Bridges are optional components of a broker system; you may choose to include them or not, depending on your overall solution architecture.

Exploring the effects of broker architecture

The Broker pattern can be very useful for structuring your solution. Like most patterns, though, it also has some liabilities.

Benefits

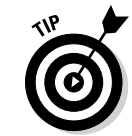
Here are the benefits of using a broker architecture:

- **Servers are invisible to clients.** The broker locates servers by using a unique identifier and makes sure that messages flow between server and client. An individual client doesn't know where the server is.
- **Client/server separation is easy to maintain as long as the interfaces remain the same.** As long as the interfaces remain unchanged, you can change and replace clients or servers (or both) independent of what happens with the other component. Changing communication paths or APIs can require you to recompile programs or reestablish registered connections.
- **Portability is enhanced.** The broker system hides underlying operating-system and network details from both clients and servers by abstracting them into the API or ABI. Using a layered architecture (see Chapter 9) helps make the servers easier to port because of the encapsulation of responsibilities within the layers.
- **Different broker systems can interoperate easily as long as they use a common protocol to exchange their messages.** Bridges (see the implementation section ["Step 5: Design the broker"] later in this chapter) make it possible for broker systems to talk to one another and to pass requests from one network to another.
- **Components are reusable because they have a clean, clear interface.** This interface makes them independent of underlying changes, as noted in the portability item earlier in this list.

Liabilities

You need to consider the following drawbacks as well as the benefits when you design a broker system:

- **Overall system performance will not be as high as that of a system with direct client/server connections.** This reduction in efficiency must be balanced by the ease of creating new services and by the portability, flexibility, and changeability of the broker architecture. In some environments, such as financial services, the broker architecture may not meet the performance requirements. The direct-communications broker variant (refer to the sidebar "Going for broker variations," earlier in this chapter) improves performance by allowing direct communication between client and server.
- **The broker introduces a single point of failure into the architecture.** All messages must flow through the broker, so if the broker is unavailable, the entire service is unavailable.



You can mitigate this problem, however, in three ways:

- By using the direct-communication variant of the broker.
- By using a process watchdog to restart the broke broker.

  You can use open source watchdogs like upstart (`http://upstart.ubuntu.com`).
- By replicating the broker and providing a way for one broker to hand its workload over to another broker. If mitigated through replication and workload hand-over, you'll also need to build detection mechanisms to know when to hand over the workload.

For more information about increasing the reliability of a broker system, see my book *Patterns for Fault Tolerant Software* (Wiley).

- **Testing and debugging are both easier and harder in a broker architecture.** They're easier because you can test the building blocks of client, server, and broker individually, using their application specifications. Testing and debugging are also more difficult because more components are involved in providing the service, which makes isolating problems harder.

Following the flow of broker messages

In this section, I want to remind you of the message sequences of common actions, because the Broker pattern is so much about the flow of information between elements. Three scenarios are most important:

- **Registration message flow:** In this scenario (refer to Figure 12-5, earlier in this chapter), the server sends a registration request to the broker. The broker records the server in its registry of service registrations and acknowledges the request.
- **Request-for-service message flow:** In this scenario (refer to Figure 12-6, earlier in this chapter), the client sends a request to the broker. The broker uses its repository to locate the server that can process the request and then forwards the request to the server. When the server replies after running its service, the broker passes the reply message back to the requesting client.
- **Bridging message flow:** When a request arrives at Broker A, the broker determines that no local servers can process the request, but Broker B has the required service. Broker A forwards the message to Bridge A, which determines that the message is destined for Bridge B and forwards the message there. Bridge B receives the request and forwards it to Broker B, which processes the message as though it were from a local client. When the server replies, the reply message follows the reverse path through the two bridges and both brokers back to the client. Figure 12-13 illustrates this scenario.

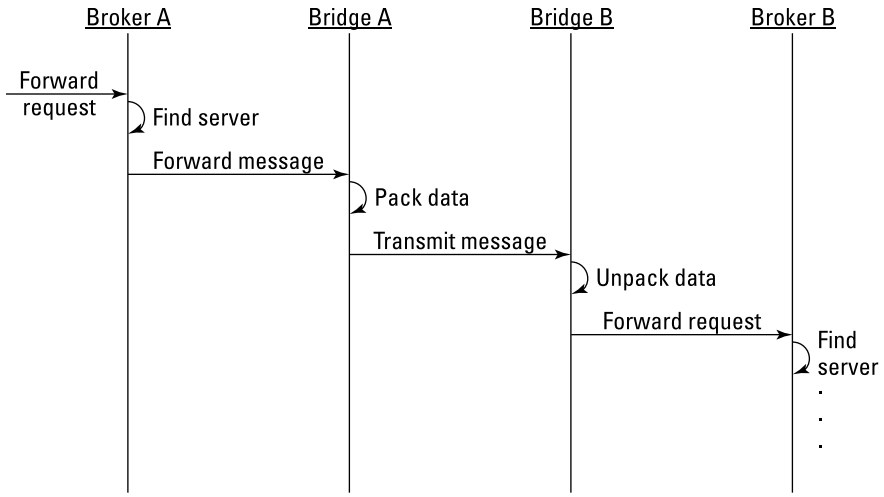


Figure 12-13: Following a bridging message flow.

Implementing a broker architecture

This section describes the process for implementing a broker system.

While you're working on steps 3 and 4, you also can work on designing the actual broker component in Step 5. By working on these steps together, you can integrate your solution by tailoring the brokers to the APIs.

Step 1: Define the object model

The broker encapsulates the interaction between the client and the server. It makes sense to refer to it as an *object model,* even if you're not going to develop actual object-oriented software. So, in this step, you need to define the rules of the system's object model, which includes providing definitions of the state of server objects and definitions of methods, as well as defining how methods are selected for execution and how server objects are generated and destroyed.

The object model must specify object names, objects, requests, values, exceptions, supported types, type extensions, interfaces, and operations. Prepare for future extensions by making the object model general enough to support these extensions.

The server state should not be directly accessible to the client. A benefit of the broker architecture is the decoupling between client and server that allows them to be changed and upgraded independently. Sharing the state eliminates this benefit.

Separating interfaces and server implementation is called *remoting*. For more information about remoting, see *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware,* by Markus Völter, Michael Kircher, and Uwe Zdun (Wiley).

Step 2: Decide the level of interoperability

Will your broker system use an API at the source-code level or an ABI at the binary level? The API provides greater flexibility, but the ABI may provide higher performance (albeit at the cost of requiring the same ABI to be used for all services).

An ABI needs support from a programming language to create the needed interface call methods. An example of this capability is in Microsoft's COM Automation software. The ABI approach gives clients direct pointers to the methods and services being invoked. The infrastructure that supports an ABI needs tables that link the pointers to the methods, both to provide the registration functionality and to provide for indirection between client and server. Every concept (what types are supported, how servers are created, and so on) in your object model from Step 1 must be represented in the ABI.

Using API implies that you have an interface definition language (IDL) to define the interface. If you use an IDL to support an API, you can map the concepts from Step 1 to programming-language-specific concepts. You can create an IDL for multiple programming languages, making the API and IDL more flexible than in the ABI approach. The IDL is compiled by a special compiler that you build in Step 6, later in this chapter. The output from the IDL compiler is either code that you include before compiling your program or a binary that you link into your program. The compiler produces two pieces of source (or binary) code — one for the client and another for the server. Both pieces are required to make the communications work.

The broker may use some of the information from the IDL to maintain information about the servers, so the broker needs to include the generated code as well.

Step 3: Specify the APIs

Define the API (or ABI) that the broker is going to provide to the clients and servers. The client must be able to build requests, pass them to the broker, and receive responses from the broker.

In this step, you must decide whether the linkages between the clients and the servers are static or dynamic. Static linkages are made between the client and server at compile time and are simpler to implement than the dynamic option. To support dynamic invocations, the broker needs to maintain the registry of servers; as a result, the API must be larger than it needs to be for static linkages. The scenario shown in Figure 12-5, earlier in this chapter, assumes dynamic linkage.

When dynamic linkages are supported, the broker must maintain the registry that can be accessed during runtime so that lookups can find servers and register new servers. The registry can be an external file that the server can access independently and examine when it needs to look up a service, or it can be internal to the broker. If you choose the internal route, the broker needs an API for the servers to register. In either case, the broker must be able to generate unique identifiers for the servers. These identifiers are the method the broker uses to send client requests to the server.

Step 4: Hide implementation details from clients and servers with proxies

In this step, you add any proxy components to the system. To the client, the proxy plays the role of a server. To a server, a proxy plays the role of a client. The proxies hide implementation details by translating requests from the client vocabulary and object model into the vocabulary and object model used by the servers.

Not every system needs proxies. Although proxies make it easier to integrate different clients and servers into the system, they also become other parts of the system that you need to maintain.

If you use the API approach with an IDL (refer to "Step 2: Decide the level of interoperability," earlier in this chapter), the proxy can be easy to implement because the IDL compiler can generate it automatically. If you choose the ABI approach, the proxies can be created and deleted by the ABI code.

Step 5: Design the broker

This step defines the broker that passes every message from client to server and back.

You can design and build different kinds of brokers in this step. See the sidebar "Going for broker variations," earlier in this chapter, for a few examples of broker styles.

To design the broker, follow these steps:

1. **Define the detailed protocol for interactions with client- and server-side proxies.**

   You must map the details of the requests, the responses, and any possible exceptions to the messaging protocol being used. These details include parameter values, method names, and return values.

2. **Define the bridge components needed to route messages to other brokers.**

   Local brokers must be available to all clients and servers within a local network. When requests need to move to other networks, you need bridge components. The bridges hide details about the distant broker from the local broker and allow the two broker systems to exchange requests and replies.

   Brokers also must maintain a registry to locate remote brokers or bridges. The requests themselves can include routing information encoded as part of server or client identifiers to simplify locating distant brokers.

3. **Define the mechanisms for exchanging requests and replies.**

   The broker must remember which client sent a request so that it can send the response from the server to the correct client. In the direct-communications broker variant, the broker isn't involved directly, so messages flow directly between client and server (or between client proxy and server proxy).

4. **Design marshaling and unmarshaling into the broker if the proxies don't contain these capabilities.**

5. **Design message buffers into the broker if communication between clients and servers is going to be done asynchronously.**

6. **If necessary, design a directory service to map local server identifiers with the physical locations of corresponding servers in the broker.**

   A simple, easy option is to use an IP port number as the directory index.

7. **Design a name server if the architecture requires unique identifiers to be generated dynamically.**

   The broker or name server will generate new names dynamically.

8. **If you choose dynamic method invocation (refer to "Step 3: Specify the APIs," earlier in this chapter), create a type registry.**

   The broker needs to maintain information about the data types that servers expect. The client may ask for this type information to help the client construct its request dynamically.

9. **Design provisions for error handling in the broker.**

   If you don't handle errors in a systematic way, debugging the system is difficult: You won't know whether the error is in your code or in your client/server interactions. In distributed systems, errors can occur at two levels, both of which must be handled by the broker:

- The servers may encounter errors (the same scenario as in a non-distributed system).
- Communication failures may occur (an error type unique to distributed systems).

You need to define what the broker should do when communication fails. Sometimes, you want the broker to resend messages until they succeed (the *at-least-once* semantic). In other cases, such as financial transactions, the risk of duplicate actions by the server is too high, so the message shouldn't be resent, or the server should recognize that the action is a duplicate and not execute it (the *at-most-once* semantic).

Another error scenario that the broker must handle is when a client requests service from a server that isn't present or that the client isn't allowed to access.

Step 6: Develop IDL compilers

If you're going to use an IDL to define the server interfaces, you should build a compiler for each language you may use. This compiler translates server-interface definitions into programming-language code.

If your system uses many programming languages, make the IDL compiler a framework that lets developers add their own code generators.

Another alternative is to use one of the open source IDL generators.

Travel agents brokering travel-related services

Travel agents are brokers for travel-related services. They take requests for clients to make reservations or obtain information from airlines, railways, cruise lines, hotels, and rental cars. The broker has the contacts and is able to navigate through a world of providers unknown to the client.

The travel providers are the servers. The broker hides the real location from the customer, which makes it easy to substitute different providers; the interfaces to the clients don't change. The systems and languages used by the broker to talk to the providers are hidden from the client, who doesn't need to know anything about those details.

Chapter 13

Structuring Your Interactive Application with Model-View-Controller

In This Chapter

- Decoupling a user interface from its data
- Designing a flexible interactive system
- Implementing a Model-View-Controller architecture

This chapter presents a really cool way of structuring your architecture when you have some data and want to look at it in multiple ways. The Model-View-Controller pattern (usually abbreviated MVC, as I do in this book) is the foundation of many systems in the real world that need exactly this functionality.

Many people associate MVC with the Smalltalk programming language, which is the best-known example. As you see in this chapter, though, MVC isn't specific to Smalltalk.

Problem: Looking at Data in Many Ways

To help you understand MVC more deeply, I walk you through an example problem in this section. You've been asked to implement a system to help some wildlife researchers understand their subjects: coyotes living in urban and suburban environments. These researchers have collected a great deal of data over many years of studying their animal subjects, including all the following information (and more):

- Birth and death records
  - Dates of birth and death
  - Identification and cross-references to siblings and parents
  - Cross-references to places of birth and death
- Location information
  - Overall territory
  - Den location
  - Birthplace and death location
  - General range and travel patterns
- Family grouping information
  - Parents, children, siblings
  - Mate
- Encounters with people
  - Nuisance reports
  - Sightings
  - Missing-pet reports within coyote territories
- Population data
  - General census information
  - Population trends (fertility rate, infant mortality rate, and so on)

Your researcher clients have asked you to prepare a display system that they can use to examine their data. They want to use some general views that they already have but imagine that they haven't thought of all possible useful views, so the system you build must be extendable. The clients want a new graphical user interface (GUI) that allows them to select what they view and to control the system.

Pondering what you need

As you sort out the data, you think about the main parts of the system you need to build:

- **Data:** The data is a primary component. You're familiar with the way that the clients stored the data (in a simple open-source database program), so I won't spend any time talking about the low-level details of data storage. Some of the data that the clients want to visualize,

however, isn't stored directly in the database; it's computed from other data in the database. This computational capability is built into components that sit right on top of the database.

- **UI control component:** Another part of the system is the user interface (UI) control component. This part interacts with the user, taking information about the data and the data format that the user wants to see.
- **Views:** You know about a few of the displays that the scientists will want to see, and you design them to be separate components of the system. All the displays are very similar, so you collect them in a third part of the system: the views.

Viewing the system flexibly

During development, the scientists ask you about getting a new view. The system already has a view that overlays coyote ranges on a map of the region; now the clients want to add data points to the view to show where the coyote dens are located and where coyotes have been interacting with pets. Figure 13-1 shows the current view that you need to extend.

The data already exists in the data store. You need to change the display controller to access this new view, which is an easy change. All you have to add is an item in a menu.

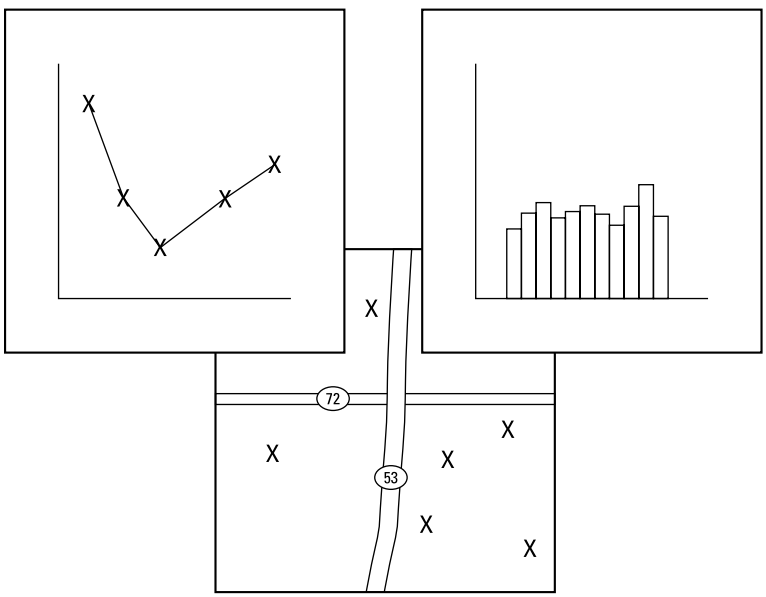


Figure 13-1: Multiple views being combined in the system.

The changes occur in the view components. You can extend the previous view by adding new lookups and plotting data.

Had the clients asked for a more-extensive change, such as adding a new kind of display, you might have added a new view component. Even this change would be straightforward, however. The views aren't integrated with the data; instead, they sit separate from it, accessing the data through an internal interface. Any interaction that the new view needs with the UI component also is simplified because of the UI's internal application programming interface (API). To start using the new view, you must make two things happen:

- The UI needs to know that the new view exists to make it available to the user.
- The new view module must register with the data store to receive notification of data updates (see the next section).

Keeping the views current

While you're working on the UI and views for the system, new data continues to arrive. New coyote puppies are born; an elderly coyote that the researchers have been following for years has died; and the scientists have been tracking several new coyotes through their radio collars. As new data arrives, the scientists enter it into the urban-coyote data model, and that data needs to appear in the views.

To keep the views updated with new data in an MVC architecture, you use the Observer pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley Professional]) or the Publisher-Subscriber pattern. (For a complete description of Publisher-Subscriber, turn to Chapter 21.) Using Publisher-Subscriber the data model assumes the role of publisher. The data model publishes changes to its data to all the other components that have registered as subscribers to that particular data, and the views become subscribers.

Changing the user interface

Although many changes are going on out in the studied population, closer to your desk, the scientists want to change the interface. First, they want to add a new way of interacting with the whole system, which is easy. Also, they want to move the whole system from the early implementation, which was driven by a main view with a menu list. Selections from this menu caused the resulting data to be displayed. Now the clients want a new icon-based system that allows them to drag data sets to display views and drop-down menus to add refinements. Figure 13-2 shows these two UI styles.

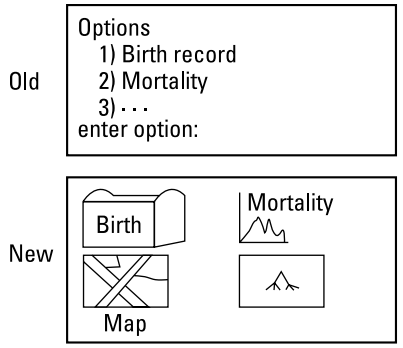


Figure 13-2: The old and new views of the system.

To make this change, you need to change the topmost structure of the UI, which results in changes in the top views and UI. All the other views, including the data display views, don't need to change. How they fit into the hierarchy of views changes slightly to reflect the change in the top-level view.

Solution: Building a Model-View-Controller System

To build the solution to the problem in the previous section, divide your interactive application into three components:

- A model that contains the data and core functionality
- Views that display information to the user
- Controllers that handle user input and tell the views what the user wants to display

Crediting MVC's inventor

A funny thing about most pattern authors and other members of the pattern community is that they don't want to take credit for something that they didn't invent. The Model-View-Controller pattern is a perfect example of this phenomenon. The concepts behind this pattern were developed by Trygve Reenskaug, but many people first saw them in the Smalltalk language, and others saw them in *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley) or in *Patterns of Enterprise Application Architecture,* by Martin Fowler (Addison-Wesley Professional). If you're curious, the original report about MVC that Reenskaug wrote for Xerox Palo Alto Research Center (PARC) is on his website at `http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf`.

Exploring the effects of MVC

The MVC pattern has been used many times to structure interactive systems. The architects who have used it have seen the benefits and liabilities listed in the following sections.

Benefits

The flexibility of decoupling the data (model) from the output (view) and input (controller) is the primary benefit of the MVC architecture. You'll see this over and over again in the following list of benefits:

- **The model is strictly separated from the UI components in this architecture.** You can use the same data to supply multiple views.
- **Changes to the data in the underlying model are reflected in all the views automatically.** This is possible because there is a single source of the data being displayed.
- **You can change the view and controller elements of the system without changing the data model.** This capability increases the flexibility of the system. You can keep the underlying model element consistent and intact, and exchange the view and controller components of the system.
- **Because the UI code is independent of the model, when you need to make major changes in the UI section, the underlying data doesn't need to change.** This kind of major change can result from moving the system to new hardware or to hardware that has a different look and feel.
- **The views don't interact.** As a result, you can change an individual view without having to make changes in the other views.
- **MVC architectures can be used as frameworks to be used and extended in other situations.** The three components are related yet independent, which simplifies maintenance and evolution.

Liabilities

In addition to the benefits, liabilities come along with using MVC. You must balance these liabilities with the benefits when you design your system:

- **Complexity is increased by separating the three components of MVC.** You have more components to build and maintain than you would if you'd designed the system as a monolith. Unless you need flexibility in the UI or the views, MVC may add more overhead than your application really needs.
- **Changes to the model are published to all the views that subscribed for them.** The number of recipients of change-notification messages increases as the system gets bigger. To overcome this liability, consider the big picture when you design the scope of changes that result in

update publications. I talk more about this drawback in the implementation section ("Step 2: Build the change-propagation mechanism") later in this chapter.

- **The controller and the views grow closer with time.** Even though the components are individuals, they have strong relationships that limit your ability to reuse one component without the others. As the system grows and evolves, views will be added to the system, along with resulting enhancements to the controller that allow the views to be selected and controlled. Because the components are so intertwined, reusing only the controller or only the views is more difficult than reusing the controller and views together. The relationship also limits your ability to insert new versions of either component, because the new version must be adapted to support the component that isn't being replaced.
- **The controller and view components know quite a lot about the model.** Changes in the model may require changes in both of the other two components. Adding indirection helps mitigate this liability, however, as I discuss in the implementation section ("Step 4: Design and build the controllers") later in this chapter.
- **Inefficient data access can result because of the separation of views and model data and the need to go through the model's API.** This problem is especially apparent if the view must request unchanged data from the model frequently. You can improve responsiveness, however, by designing the view to cache data.
- **Both controller and view components require changes when they're ported to a new system.** These components contain some platform-dependent code, so when the components are ported to a new system, the platform-dependent code requires changes.
- **MVC arose before modern UI tools were created.** MVC is useful for increasing portability. If portability isn't a critical requirement, the use of a UI toolkit can be a more appropriate overall solution. These two solutions are incompatible because toolkits that specialize in creating UIs include their own flow of control and their own mechanisms for accessing the model, whereas the MVC controllers want to control the way that user interaction occurs.

  Suppose that one or another controller component wants to use pop-up windows or manage window scrolling. You can use wrappers to connect the MVC components with the UI toolkit components, but this solution is complicated and hard to maintain over time. Another problem is that the toolkit may expect to interact with the model in ways that are incompatible with the MVC's controller component. Yet another closely related problem is the contention between the MVC's controller and the toolkit's control components over the processing of events and callbacks to the user.

When you face any of these problems, consider eliminating the MVC's controller and using the controller provided as part of the toolkit.

Inspecting MVC's moving parts

So far, I've talked about the parts of MVC only in general terms. In this section, I explain the roles and responsibilities of the three parts of MVC: the model, the views, and the controllers.

The model component

The model component contains the core of the application — both the application's data and the important data-related functionality. The model provides procedures and methods to access the data. These procedures and methods are called by the controller in response to user control. The model also provides functions to access the data stored in the model that the views need to construct their displays.

Figure 13-3 shows the Class-Responsibility-Collaborator (CRC) card for the model component. (For a reminder on CRC cards, check out Chapter 2.)

Class Model	***Collaborators*** • View • Controller
Responsibilities • Provide functional core of an application • Register views and controller interest in model data • Notify registered components about data changes	

Figure 13-3: The model CRC card.

The model must keep the data that it stores up to date, so it must have mechanisms to update the data internally and to report the updates to all the views that are interested in that data. Frequently, this change-propagation mechanism is implemented via the Publisher-Subscriber pattern, which I introduce in Chapter 21.

A variation of the model is for it to remain passive and not publish updates. In this variant, the views and controllers ask the model for updates rather than subscribing and waiting for updates.

The view component(s)

Information in the model is displayed for the user through the view component. A system can (and usually does) have more than one view component.

Each view provides the user different ways to visualize the data. The views receive updated data from the model by subscribing to the model's publication of changes. When updated data is received, all the views update what they're showing the user.

Figure 13-4 shows the responsibilities of a view component.

Class View	***Collaborators*** • Controller • Model
Responsibilities • Creates and initializes its controller • Displays information to the user • Updates itself when new data arrives from model • Retrieves data from the model	

Figure 13-4: The view CRC card.

During initialization, all the views register with the model's publication process, ensuring that the views have up-to-date data.

A one-to-one relationship exists between views and controllers — that is, each view has a controller. Each view also may have subviews. In a typical application, buttons, scrollbars, and menus all are subviews. A hierarchy of views and controllers provides the displays and behaviors that you expect to see.

In general, the best design is to put the view component in charge of the creation of the controller component. The view component frequently offers the controller some functionality for manipulating the display. This functionality is used for display changes that affect only the view data and not the model data, such as scrolling.

The controller component

The controller interacts with the user and processes user inputs as events. When events arrive, the controller checks to see whether the event applies to it; if it does, the controller processes the event. If the event isn't relevant to the controller, the controller takes no action. The scrollbar controller does nothing when the mouse clicks a button, for example.

Figure 13-5 shows the CRC card for a controller.

Class Controller	*Collaborators* • View • Model
Responsibilities • Accepts user input as events • Translates events into requests for the model or display request for the view • Updates itself when new data arrives from the Model	

Figure 13-5: The controller CRC card.

Sometimes, the controller's behavior depends on the model's state. In such a case, the controller must register with the model's change-propagation method, just as the views do. This registration is required when the presence of certain data in the model may allow the creation of new menu items, for example.

Views can have more than one controller. Some elements of the screen can be edited and others can't, for example. In such a case, you can put the controls for these elements in separate controllers.

Implementing MVC

To implement MVC, execute the following steps. The first six steps are fundamental when you're designing your application to use MVC; the remaining steps help you refine your use of MVC to make it more flexible.

Step 1: Separate the core functionality from the UI behavior

Analyze the application domain of the problem you're solving, and answer the following questions:

- What are the core data parts?
- What computational functions are performed on the data?
- What is the system's desired input?
- What is the system's desired output?

The input and output go into the view and controller components designed in steps 3 and 4. In Step 1, you design the model component to store the data

and perform the core computational functions. Design functions to access the data that the views will use. Also, decide what data and functionality the controllers and views should be able to access directly, and define the access interface.

In the coyote-study example, you decide to reuse the existing data schema and to build the model with a few helper classes to perform the computations.

Step 2: Build the change-propagation mechanism

The model component is the publisher from the Publisher-Subscriber pattern (see Chapter 21). Design the registry that the model will use to remember which views and controllers have subscribed to the data. Figure 13-6 shows an example of a registry. You also need to design the procedures that the views and controllers must use to become subscribed in the registry or to unsubscribe themselves.

Data	Subscriber
Birth Record	View 3
Location Data	View 1
	View 2
	View 8
Relationships-birth	View 4
Relationships-mate	View 6
.	
.	
.	

Figure 13-6: The model's registry of subscribers.

The model's publication of updated data should call the update procedure of all the subscribed views and controllers. Everything that happens in the model that changes any of the data must call the publication mechanism to push the change out to the subscribers.

For times when the data in the model isn't changing and the views or controllers need access, you should build a separate access mechanism. This mechanism allows the view and controller to request the current state of the data when they start.

Step 3: Design and implement the views

Each view presents the data from the model to the user in a different way. In this step, you design and build the procedures that get the data from the model and then display it.

For each view, you must decide on its appearance and then create the required drawing software to display it. The drawing software accesses the data it needs from the model through the access routines defined in Step 2. The actual drawing is done in conjunction with the graphical interfaces available in the platform being used.

When data in the model is updated, the changes are propagated to the views that subscribed for notification. The views need to register with the model for the data they need.

Different views may need different data from the model. You need to include in the view the necessary update functionality so that when an update is received from the model, the view reflects it in the display.

The easiest way to handle this requirement is to retrigger the drawing procedure with the new data. If the display is complex, however, this solution becomes inefficient. You have several ways to compensate for this inefficiency:

- **Extra information from the model gives the view information about the scope of the change.** If the change is minor, the view can call some other display-update method that doesn't require a complete redraw of the data.
- **Wait until a flurry of updates has been received and the view determines that no more updates are anticipated.** Two methods are available to the view:
  - The view can use timers between updates to see when a minimum stable threshold is exceeded, after which it's safe to redraw the view.
  - The view can set a timer, and when the timer fires the view, the display is redrawn. The duration of the timer is set to be slightly longer than the duration of a typical flurry of update requests.

I only hint earlier that views must be initialized whenever they're created. The view must subscribe to the model for whatever data it needs for its display. The view also must instantiate its relationship with the controller. (For more about this relationship, see "Step 5: Build the relationship between the view and controller," later in this chapter.)

In your coyote-data display system, consider the view that shows the geographic ranges of the coyotes under study and the view that shows the pedigrees of individual animals. Each type of view requires different information from the model (see Figure 13-7). During initialization, subscriptions are made with the model for the data that each view needs.

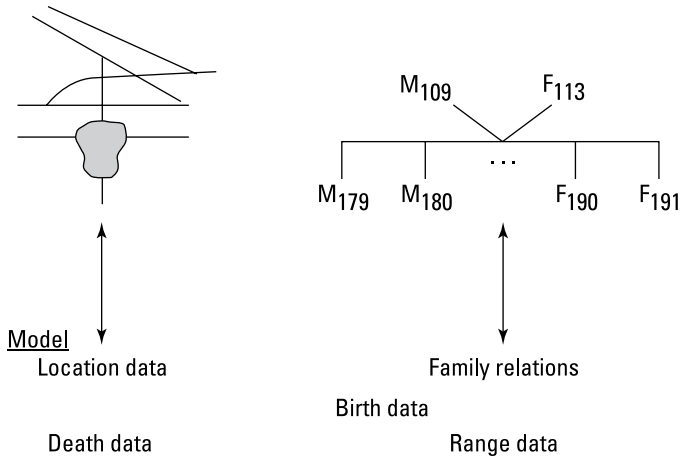


Figure 13-7: Two views mapping to different data in the model.

Step 4: Design and build the controllers

Every view has a controller, and the complete system has multiple controllers. Each view controller receives the events that contain UI instructions, interprets those instructions, and passes control information off to the view with which it interacts. (The event-processing mechanisms are assumed to be available in your operating system and aren't part of the MVC pattern.) In Step 6, you set up overall event processing and kick things off.

The behavior of the controller can depend on the state of the model. Depending on the model, only certain control capabilities may be present. Because the scientists using the coyote data want the population-by-zone data to be accurate at all times, there may be a time when the controller won't allow the user to request an update — perhaps the mapping to zones hasn't been recomputed after a scientist updated the model. Figure 13-8 shows an example.

The controller component is linked to a model and to a view during the controller's initialization. As part of the initialization, the controller registers as a subscriber for any data that it needs to control its actions.

A close relationship exists between the controller and the computational core of the model. This relationship is a problem when you want the controller to be reusable with other models. You should use the Command Processor design pattern (see Chapter 20) to add some indirection and isolate the controller from the model. In that pattern, the model takes the role of supplier, and the command and command processor components are between the MVC's model and controller.

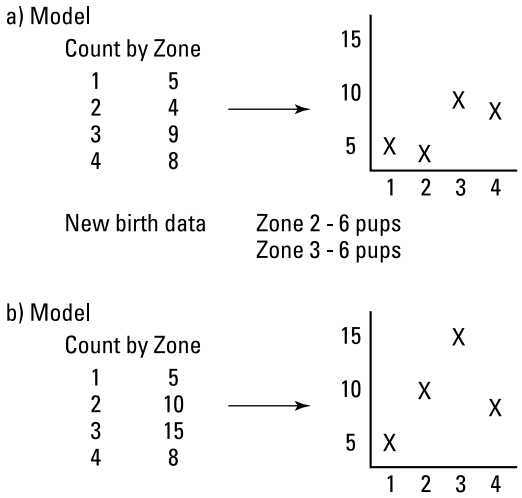


Figure 13-8: View behavior changing with model state.

Step 5: Build the relationship between the view and controller

Each view has a special relationship with its controller that needs to be created during their initialization. Within the view class, you should define a `makeController()` method to create the controller if it isn't included in your program's initialization code.

Step 6: Get the MVC started

With multiple views and controllers, you need to tie all the elements together and get them started — preferably in an external place such as a main program.

The controllers in MVC rely on events being passed their way. They respond to these events to cause the views or the model to be changed. An important detail in starting the MVC is starting the event processing, because the event-handling mechanisms aren't explicit parts of the MVC pattern.

Steps 1 through 6 build the basics of MVC. If you want your MVC implementation to be more flexible and extendable, proceed to the remaining steps.

Step 7: Create dynamic views

If your application may open and close views during execution, it's helpful to have a component to manage the active views. Use the View-Handler pattern (see Chapter 20) to structure this component. One capability that this component can provide is to terminate the application when the last view is closed.

Figure 13-9 shows the CRC card for the View Manager component.

Class View Manager	***Collaborators*** • View
Responsibilities • Opens, manipulates, and destroys views	

Figure 13-9: The View Manager for an MVC CRC card.

Step 8: Create changeable controllers

In any application, the design of the model is static. You can add and remove views to show different aspects of the model or to accommodate different display devices, but the views are still relatively stable.

Because the controller in an MVC architecture is separated from the views and the model, you can make your controllers changeable or pluggable. The controllers must adapt to the interfaces that you designed in steps 2, 3, and 5, of course, to work correctly with the views and models in your application.

Pluggable controllers allow your application to adapt to new input devices without changing the views or underlying model. You could use different controllers to reflect the unique needs of novice or expert users, for example. Yet another use is to create controllers that accept only limited input, thus providing what is essentially a read-only view.

Make your MVC into a framework

The primary benefit of MVC is the separation of the parts, which allows you to change the parts. In Step 8, I provide some ideas about how you can interchange controllers to provide a different kind of interface for your application. Earlier steps describe the ease of supplying different views.

Because the parts of your MVC are portable and adaptable, they're useful packaged as a framework, which makes it easy to reuse the components in other applications.

Step 9: Design the infrastructure for hierarchical views and controllers

Screen elements such as buttons, scrollbars, and menus are basic building blocks that are common to many views. You build a UI by composing these predefined parts, each of which is implemented as a view.

Figure 13-10 shows a class hierarchy of these composed views, using the Composite pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*).

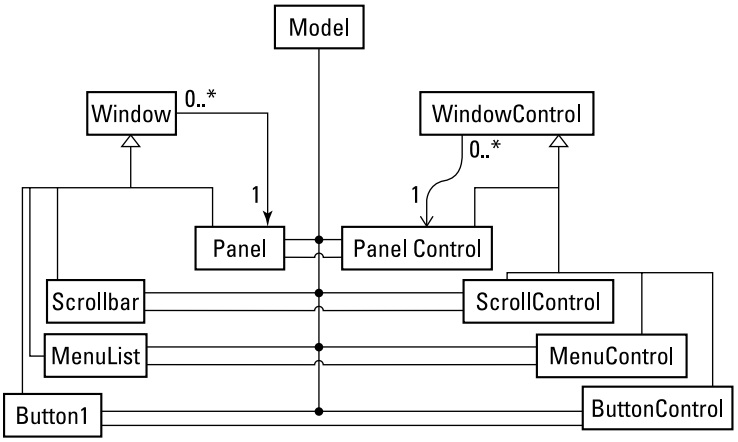


Figure 13-10: Composing views into a hierarchy.

Event handling with this composed display can become a problem, however, because knowing which controller should handle the event may not be obvious. When a button is clicked the event handler, not knowing which controller to send the event to, will send it to all the controllers. The `ScrollControl` should ignore it, as should the `MenuControl`; the `ButtonControl` should process the event.

One way to solve this problem is to use the Chain of Responsibility pattern (from *Design Patterns*). When you apply this pattern, each controller that receives the event either processes the event (if appropriate) or passes it to another controller. The other controller is associated with either its parent view (from `ScrollControl` to `MenuControl` to `ButtonControl`, for example) or a sibling view.

Step 10: Remove system dependencies

It's easy to accidentally let your controller and views become dependent on the specifics of the host system. When you're building only one instance of

a solution, this dependency isn't always a problem, but dependencies make things more difficult if you want to reuse the controllers and views.

To achieve a clean separation of your controllers and views, you can introduce other classes for them that encapsulate the hardware specifics, allowing the view and controller classes to access them indirectly. To provide this indirection, use the Bridge pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*) to separate hardware abstractions from hardware specifics.

A pair of *display* classes is created to support the views. An abstract class provides the methods for common tasks such as drawing lines, displaying text, creating windows, and changing the appearance of the mouse. A concrete class (or classes) is created to implement these tasks in a host-dependent manner, calling the appropriate host-specific libraries and functions to achieve the desired results.

Abstract and concrete *sensor* classes do the same for user input. The abstract sensor class provides generic, host-independent methods, and the concrete class invokes the host-specific capabilities.

Designing the display and sensor classes (see Figure 13-11) can be hard. One decision that you need to make when designing them is how abstract they should be:

- At one end of the spectrum, the display and sensors are built with minimal common functionality. Only methods that appear on all host platforms are created.
- At the other end of the spectrum, the abstract display and sensor classes offer high-level abstractions of the capabilities. These classes thereby offer capabilities that build on the basic, common functionality of all hosts.

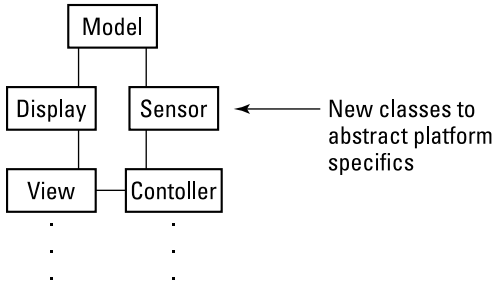


Figure 13-11: Insert display and sensor classes to hide system dependencies.

Controlling the view of a football game

When televising a football game, the television network has multiple cameras all focused on the action. The director chooses which camera angle to broadcast to the TV viewers based upon the play on the field. There's only one game being captured by a group of cameras — the game corresponds to one model in the Model-View-Controller pattern in this chapter. The cameras provide the views and are given instructions about what to zoom in on by the director, who is the controller. The camera crews all have different views of the play and are following the play all the time.

The first approach leads to applications that look the most similar across platforms. The second approach helps the application match platform-specific characteristics and guidelines better.

Figure 13-12 shows CRC cards for the display and sensor classes.

Class Display	***Collaborators***
Responsibilities	• Views
• Create windows	• Model
• Draw lines and text	
• Change cursor	
• Pass update requests to views	

Class Sensor	***Collaborators***
Responsibilities	• Controllers
• Abstract events	• Model
• Translate events for different	
• Pass update requests to controllers	

Figure 13-12: Display- and sensor-class CRC cards.

Seeing Other Ways to Manage Displays

Several other patterns are similar to MVC. In this section, I tell you about one variant of MVC and also give you insight into how MVC, as discussed in this chapter, differs from the Presentation-Abstraction-Control (PAC) pattern, which is the topic of Chapter 14.

Combining controller and view

Document-View is a variant of MVC that doesn't enforce the strict separation between controller and view that exists in the MVC pattern. In some implementations, the controller and view are tightly interwoven. In the X Windows System, for example, events are dispatched to a window — which means that they're sent to a view rather than to a controller. You lose flexibility and the ability to change controllers relative to views when they're combined in Document-View.

The Document component corresponds to MVC's model component. Because Document and View are loosely coupled, the benefits associated with their separation are achieved in either the Document-View or MVC pattern.

Document-View is useful when you don't need the flexibility of separate views and controllers, or when you want an architectural model that corresponds to a particular toolkit, like the X Windows System.

Comparing Presentation-Abstraction-Control

In the next chapter, I introduce the PAC pattern — another architectural pattern for structuring interactive applications, but one that's quite a bit different from MVC.

MVC is all about creating independent components that are responsible for realizing the model, view, and controller functions discussed in this chapter. These components work together but are discrete parts of the system. There may be multiple views and controllers, but they're all roughly equivalent.

In PAC, by contrast, the architecture is created with a hierarchy of agents, each containing all three parts of the name: a presentation part, an abstraction part, and a control part. Figure 13-13 compares the two architectural patterns.

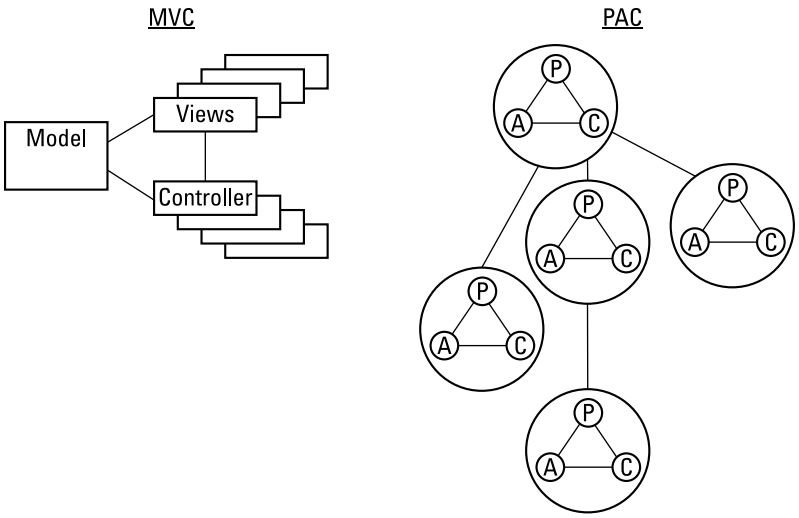


Figure 13-13: A comparison of the MVC and PAC architectures.

The most important difference between these two patterns that should lead you to choose one or the other is that PAC is built around the notion of independent components cooperating to perform the application, whereas MVC is more about the flexibility of the human–computer interaction.

Chapter 14

Layering Interactive Agents with Presentation-Abstraction-Control

In This Chapter

- Building a system of agents for an interactive application
- Assembling legacy display components
- Implementing the Presentation-Abstraction-Control architecture

In this chapter, I tell you about Presentation-Abstraction-Control (PAC), a pattern for structuring your interactive application when the parts are autonomous components with their own independent capabilities.

I refer to these autonomous components as *agents,* and I define an *agent* as a component that has these capabilities:

- It can receive events and forward them to other agents.
- It contains data structures that store the agent's information as well as the agent's state.
- It can perform at least the following computations:
  - Processing incoming events
  - Updating its own state
  - Generating new events that are sent to other agents

Agents may range in size from a single object to something as complex as a complete software system.

Note: This definition of *agent* is specific to this chapter. If you've read about agents elsewhere, you'll see that the definitions are related, but in this chapter, I don't talk about anything more advanced than what I outline above.

Understanding PAC

At its most abstract, a PAC architecture consists of six kinds of classes: three at the agent scale and three within each agent. Figure 14-1 shows this overall view.

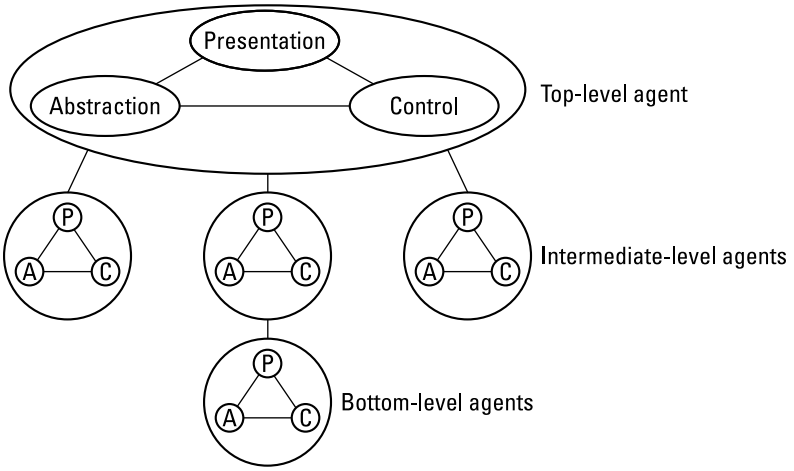


Figure 14-1: The overall hierarchy of agents and PAC within agents.

At the agent scale, the three classes are top-level, bottom-level, and intermediate-level. Figure 14-2 shows Class-Responsibility-Collaboration (CRC; refer to Chapter 2) cards for these three agent classes.

The agents in a PAC hierarchy behave like layers in the Layers pattern (see Chapter 9), in that they communicate only with agents in adjacent layers. If information needs to flow from an agent of one level to another agent of the same level, it must flow upward through an intermediate-level agent before returning to its destination.

Class Top-level Agent	***Collaborators*** • Intermediate-level Agent
Responsibilities • Provide the core functionality of the system • Control the PAC hierarchy	

Class Intermediate-level Agent	***Collaborators*** • Top-level Agent • Intermediate-level Agent
Responsibilities • Coordinate lower-level agents • Compose lower-level agents to appear as a single abstraction to higher-levels	

Class Bottom-level Agent	***Collaborators*** • Intermediate-level Agent
Responsibilities • Provide specific views of the software or service, including interaction with users	

Figure 14-2: Agent CRC cards.

Inside all the agents of a PAC architecture are three classes: presentation, abstraction, and control. Figure 14-3 shows CRC cards for these internal classes.

Class Control	***Collaborators***
Responsibilities • Provide access to services provided by this agent • Coordinate the PAC hierarchy • Maintain overall user interaction	• Control classes in higher-level agents • Control classes in lower-level agents • Presentation class in this agent • Abstraction class in this agent

Class Abstraction	***Collaborators***
Responsibilities • Store internal data used by this agent	• Control class in this agent • Presentation class in this agent

Class Presentation	***Collaborators***
Responsibilities • Provide view of abstraction to user • Provide and control user interactions	• Control class in this agent • Abstraction class in this agent

Figure 14-3: Internal PAC CRC cards.

Each of these classes inside an agent has the same responsibility relative to the others. Depending on a class's location in the overall agent hierarchy, however, the functionality of one or more of these internal classes is minimized or emphasized (see Figure 14-4). In the top-level agent, the abstraction is emphasized because it stores the model data. In the bottom-level agents, the presentation class is emphasized because it provides the lowest-level display functionality. The intermediate-level agents can fulfill several roles; as a result, the responsibilities of all three internal classes may be more balanced.

Political polling by city, state, and nation

Political parties use polls to interview the public to understand how the public feels toward their candidates before an election. Some elections are at the local city level, some elections are at the state level, and some elections are at the national level. The parties want poll results at each of these different levels. Each level's results are a combination of results at lower levels — the statewide prediction is the sum of the local predictions across the state, and the national prediction is the sum of the state predictions. The information gathered at the lowest level cascades upward to the larger bodies.

Comparing this hierarchy of polls to the Presentation-Abstraction-Control pattern, the national poll corresponds to the top-level agent; it takes the results from the intermediate-level agents, which are the states results. The bottom-level agents are the citywide polling data.

Each agent is responsible for its own results, which can be accessed and used by the levels above it. It's easy to add new intermediate levels — for example, counties that combine several cities but are still part of a state. The agents can all work independently and in parallel. Voter opinion can be collected in multiple cities simultaneously.

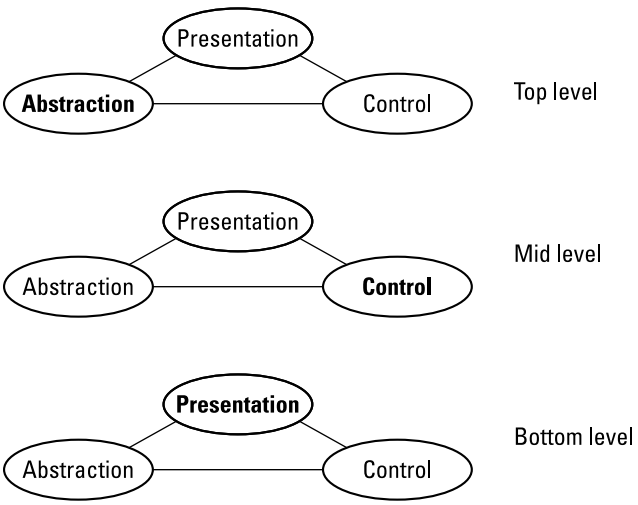


Figure 14-4: Relative importance of PAC classes in different layers of agents.

Problem: Coordinating Interactive Agents

Continuing the example introduced in Chapter 13, your problem in this chapter is to combine several applications being used in an urban-coyote

study. The scientists have been using individual programs to access different data sets and data displays; now they'd like to transition this collection of programs to a unified system. When you get this assignment, your goal is to reuse as much as possible of the existing programs. You and the scientists expect that it will be easier and faster to create new views of the data if all the existing views and data are combined into a single program.

Currently, all the individual data programs use their own private data sets. The scientists have two goals for the new application:

- They want to view all their data in one program with a simplified, unified user interface (UI).
- They want to combine all their data into one large data set. Currently, each of the different programs stores its own data, and some overlap has led to data-integrity issues in the past. The scientists think that if all the data is together and in the same place, they may see new relationships between the data sets that have been invisible to them in the past.

You also have a goal: You want the new system to be easier to maintain. Keeping all the little programs consistent and synchronized is hard.

Combining the programs

After some research and software archaeology, you find that the scientists' separate programs have similar structures. This similarity allows you to encapsulate the separate programs as objects that fit within a new hierarchy.

In addition to reusing the existing programs, you need to add some other objects to the unified application, including error handling and a new data-entry method. Your class diagram looks like Figure 14-5, in which the old display programs are labeled Program1, Program2, and Program3.

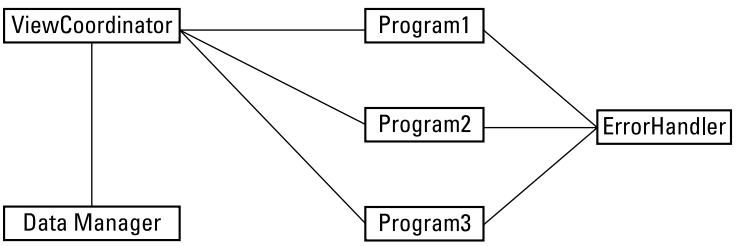


Figure 14-5: Different data displayed as objects in the solution.

The data from all the different data sets is being combined into one data repository. You pick a popular open-source database because it seems appropriate and you want to find out more about it. Regardless of the type of database or file you store the data in, you create a repository object to access the data and to provide it to the views.

Because you have this new centralized data repository, you also need to create a new way to enter the data and to access it directly. You discuss this issue with the scientists and decide that data entry should be done through a simple spreadsheetlike interface.

Each of the formerly independent programs needs some modifications that allow it to access data from the shared repository instead of from its internal data store. These modifications aren't hard, because, for the most part, the individual programs were written with maintainability in mind, and they isolated the mechanics of their local data stores from the display functions. By changing some application programming interfaces (APIs) and writing some Adapter software, you're able to redirect data access to the centralized repository. (The Adapter pattern is available in *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley Professional].)

Ruling out MVC

You may have read about Model-View-Controller (MVC) in Chapter 13, so you investigate using it for this application. Pretty quickly, however, you realize that MVC isn't appropriate, for the following reasons:

- **The model component in the MVC architecture pushes changes out to the views.** In the new program, you want the formerly separate programs to act more autonomously and to retrieve data from the repository when they need it.
- **The views and controllers in MVC are closely related.** Many times, these components are implemented as subviews. When you combine existing systems into a new unified system, each of the views in the new system was an independent program before, so it stands alone and doesn't have interfaces to other programs, like the views in a typical MVC system. Conversion of the separate programs to objects within an MVC architecture would be hard, and you wouldn't be able to reuse as much of the existing programs as you'd hoped to.

The following section provides a more general comparison of the MVC and PAC patterns.

Comparing PAC and MVC

In Chapter 13, I introduce MVC, another architectural pattern for structuring your interactive applications. MVC and PAC are quite a bit different.

MVC helps create one hierarchy of control, abstraction, and presentation components. By contrast, the PAC architecture is useful for assembling a hierarchy of separate hierarchies.

In PAC, the architecture is created with a hierarchy of agents, each containing all three parts of the name: a presentation part, an abstraction part, and a control part.

The most important difference between these two patterns that should lead you to choose one or the other is that PAC is built around the notion of independent components cooperating to perform the application, whereas MVC is more about the flexibility of the human-computer interaction within a single overall component.

Using separate agents

In the scenario I've been describing, each of the formerly separate programs is to remain autonomous. Each one behaves like a separate agent, all of which share the overall system hierarchy and a repository of data with the other agents. (For a reminder, see Figure 14-1, earlier in this chapter.)

Each of the agents in your new program once was a separate, independent program. As a result, it had its own data abstraction component, in which it structured the data for which it was responsible; its own view component, which it used to present the display to the UI; and its own control part for receiving input from the user. Figure 14-6 shows several display components with their subcomponents.

Each agent continues to have these three subcomponents: data abstraction, presentation, and control. In "Combining the programs," earlier in this chapter, I point out that the abstraction subcomponent needs to change from accessing data local to the agent to centralized data. Similarly, the presentation and control subcomponents need to be modified slightly to reflect the fact that they aren't in a stand-alone environment.

You have to add a high-level agent to control and coordinate all the separate views that once were separate programs. This agent will provide the main UI that lets the scientists pick the display they want to see.

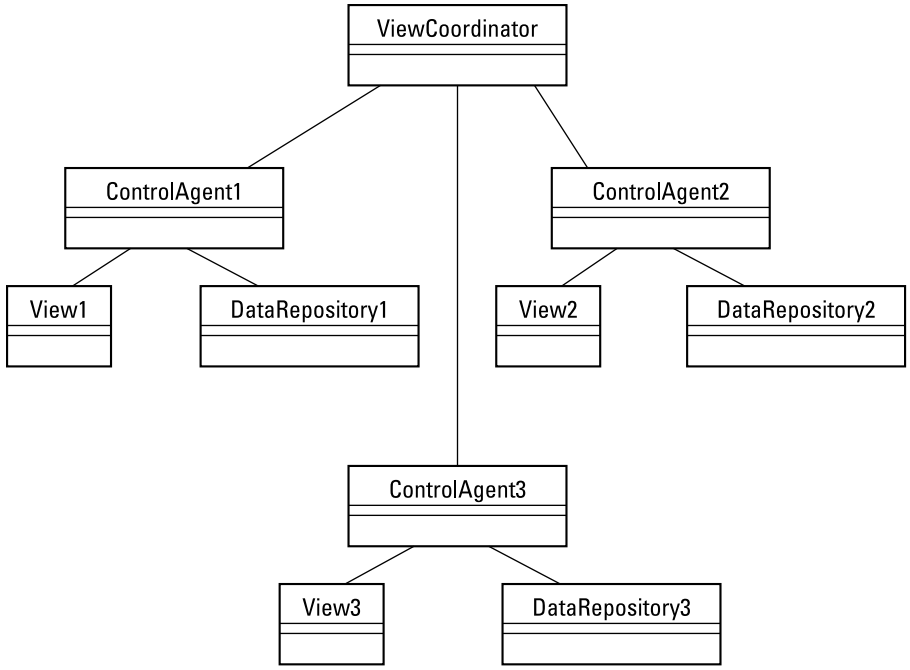


Figure 14-6: A hierarchy of components and subcomponents.

By building the system this way, you achieve your goals:

- You reuse the stand-alone programs.
- You merge the data and data handling, which is going to reduce maintenance problems for you and the scientists in the future.
- You achieve a single system to which you can easily add new displays in the future.

Solution: Creating a Hierarchy of PAC Agents

The PAC architecture defines a hierarchy of cooperating agents, each of which has responsibility for part of the application's functionality. Each agent contains three components: one for presentation, another for abstraction, and yet another for control. These three components isolate user interaction from core functionality and define cooperation among agents.

Exploring the effects of PAC

As with all patterns, certain benefits and liabilities are associated with the PAC pattern. I tell you about them in this section.

Although the benefits of PAC can be significant, the liabilities also can be significant — to the point that you shouldn't use PAC for some problems (see "Knowing when — and when not — to use PAC," later in this chapter).

Benefits

Here are the benefits of a PAC architecture:

- **PAC excels at separating concerns.** Each agent is exclusively responsible for a part of the application. It maintains its own state and data, coordinating with other agents when necessary. The boundaries between the agents are well defined and discrete. You can define and build each agent's internal characteristics as most appropriate. Among other benefits, this feature allows the agents to be developed independently, maybe by separate teams.
- **PAC supports evolution and extension.** Because each PAC agent is separate, with a well-defined interface and discrete boundaries, the changes within an agent don't affect the other agents. You can modify the internals of the agent without causing a major upheaval in the system.
- **Adding new agents to the system is easy.** You just squeeze them in between the existing agents in the system. You don't need to worry about effects on other parts of the system, because the agents are discrete and self-contained. You could add a new kind of view to the coyote-data system, for example, by adding its new agent and then modifying the small number of other agents that start and trigger the display of the new agent.

  The well-defined interface between agents means that new agents don't require interface changes. You can handle new agents appearing in the system by creating a registration interface so that new agents can register their presence; the agents that need to know about the new agents can pick up this information from the registration.
- **PAC agents are easy to distribute, which provides better performance and multitasking.** The agents can be spread out as different threads or different processes (see the sidebar "Implementing agents as processes," later in this chapter), or even on separate computers. Changing the system from one distribution model to another is simple; although you have to change the agents, the changes are confined to the agent's control component.

Liabilities

Like all patterns, PAC has some liabilities:

- **The overall complexity of the system increases if you use PAC and its agents indiscriminately.** An easy trap to fall into is using an agent for every low-level item, such as drawing a square or a border. If you do this, you'll soon be buried by all the agents! Complexity also increases in the higher-level agents because other agents must be created to coordinate the explosion of low-level agents.

  You need to think about your design carefully, consider its level of granularity, and at some point stop refining things into ever-simpler bottom-level agents.

- **Overall control of all your agents gets complex and makes the overall solution complex.** The control components mediate between the abstraction and presentation components, as well as between different agents. This mediation capability is crucial for achieving good internal collaboration; poor choices here negatively affect overall architectural quality. The external interfaces and APIs to the control component shouldn't refer to internal characteristics or naming conventions that the other agents won't know. The control component should do any translation and mapping required on information coming into or passing out from an agent's other components.

- **The complexity of PAC architectures can make them inefficient.** The communication flow within a hierarchy of agents is similar to the flow in the Layers architecture (see Chapter 9): Messages from the top of the agent hierarchy must pass through all the intermediate agents to reach the lower-level agent recipient. If the agents are distributed across a network, the networking protocols and data-transfer mechanisms further reduce efficiency.

Knowing when — and when not — to use PAC

The PAC architecture isn't ideal for every situation, and its drawbacks limit its applicability. As the scale of the concepts in the architecture shrinks, PAC becomes less and less applicable. If the modeling of each object in a graphical editor must be represented by its own PAC agent, for example, the number of agents explodes, and the overall architecture becomes much more complex. This complexity negatively affects your ability to build and maintain the system; it also negatively affects the efficiency of the resulting application.

If the abstraction of the concepts is larger, especially as the concepts start to require their own UIs, PAC becomes an appropriate pattern. In these situations, PAC provides a maintainable architecture with clear interfaces and separations of concerns among the various tasks. Figure 14-7 shows the continuum of PAC applicability.

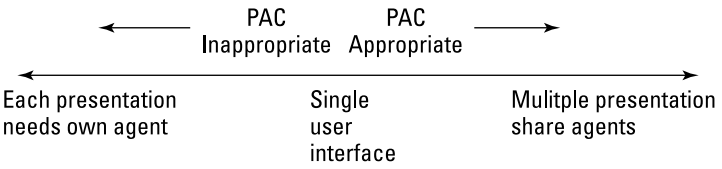


Figure 14-7: The continuum of PAC applicability.

Looking inside PAC architecture

In this section, I dig into more details on the parts of PAC. Refer to the overview of the solution and the figures in the "Understanding PAC" section as you're reading.

Top-level agent

The top-level agent provides the global data model for the system. Internally, it's the abstraction component that provides this capability. The abstraction component contains methods that allow the data to be manipulated and accessed; the data is independent of how the system displays it. In a mapping system, for example, the model is stored in terms of real units such as miles and kilometers. This independence makes the agent portable to other applications and new views.

The presentation class of the top-level agent doesn't have much to do. It may have some systemwide display functionality, or it may not do anything. The top-level presentation class is a good place to store the fundamental display elements that most of the other agents will use, such as scrollbars and borders.

The control class at the top level has three responsibilities:

- **It coordinates the PAC agent hierarchy.** As the topmost layer in the logically layered architecture, it makes sure that lower-level agents get the information and guidance they need.
- **It allows lower-level agents to use services provided by the top-level agents.** These services are mostly related to the global data model stored by the top-level agent.

- **It maintains the status of user interaction inside the system.** It may check to see whether certain operations by the user are possible in a given data model, or it may store history or permit undo/redo capabilities.

Bottom-level agents

The bottom-level agents store the key display concepts found within the semantics of the application. These concepts can be small or large, such as simple rectangles or complex maps. They can have semantic importance to the application, such as mailboxes in a network traffic management system, or they may be generic display widgets, such as scrollbars.

The concepts that the bottom-level agents are responsible for are the smallest things that the user can manipulate.

The abstraction component stores any agent-specific data. Unlike the top-level agent, though, it doesn't store data for any of the other agents in the system.

The presentation class of a bottom-level agent provides a specific view of the concept, and it provides access to all the functions that the user may apply to that concept. In the example of a map, it provides access to the panning and zooming controls. It also maintains information about the view internally, such as the current state of what the user is examining.

The control component maintains consistency between the abstraction and presentation components, acting as an interface and preventing the agents from having dependencies between the classes. Also, events and data are exchanged with higher-level agents. Incoming events are forwarded to the bottom-level agent's control class, whereas incoming data is forwarded to the bottom-level agent's presentation class. Outgoing events and data are forwarded to the higher-level agents.

In the coyote display system from Chapter 13, the old programs become the bottom-level agents.

Intermediate-level agents

The intermediate-level PAC agents coordinate the activities of other agents in the system, or they combine or compose the results from other agents. The combination and composition are done mostly by the control class.

In the coordination role, an intermediate-level agent maintains consistency among other, lower-level agents in the system. It makes sure, for example, that multiple views of the same data managed by lower-level agents are consistent.

In the composition role, an intermediate-level agent defines a new abstraction for the system. Then it pulls in the needed lower-level agent's results (or presentations) to create the instance of that abstraction that the user can see and manipulate.

The abstraction class of the intermediate-level agent manages the data needed internally. The presentation class provides any UI capability that goes along with the intermediate-level agent's role.

Implementing PAC

Implementing a PAC architecture involves the ten steps described in the following sections. You can repeat any step or group of steps as necessary throughout your implementation.

As you work through these steps, you may get confused about top and bottom roots and leaves, so Figure 14-8 shows a reference to this terminology.

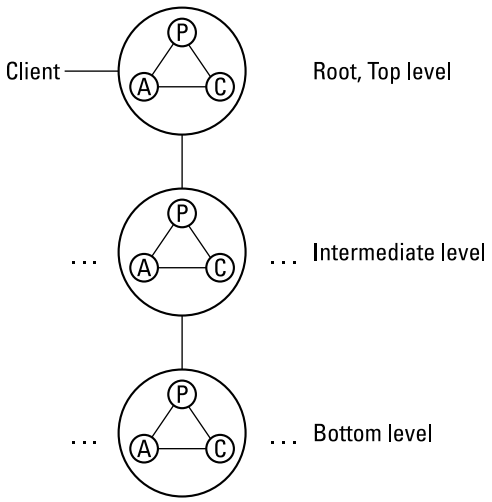


Figure 14-8: PAC tree terminology reference.

Step 1: Define the application's model

In this step, you study the problem's domain and map it to the overall software architecture, which means deciding on the best decomposition and abstraction of the problem.

Don't think about PAC agents or distribution of the components while you're working on this step. You work on those aspects in later steps.

To understand the application's model, you need to know the answers to the following questions about the system and its components:

- What are the services that the system provides?
- What components does the system need to deliver those services?
- How do the components of the system relate to one another?
- What is the collaboration between the components?
- What is the data on which each component acts?
- How does the user interact with the system?

Step 2: Develop a general PAC hierarchy

Form a general hierarchy of PAC agents. You still haven't identified all the agents — you start doing that in the next step — so don't get lost in the details yet.

A strategy that you can use to develop the hierarchy is *lowest common ancestor.* When several components rely on the services of other components, put them in a hierarchy with the source component as the root. This strategy identifies those agents that provide common services and elevates them to the top of the hierarchy.

In the coyote research project, all the different displays will be driven from one view coordinator component — which, therefore, is the root of the display hierarchy. Figure 14-9 shows both this subhierarchy root and the next one. All the views require access from the data repository. As a result, the repository has been elevated to the root of the tree, which is the topmost level in this example, as I explain in the next step.

Implementing agents as processes

Each PAC agent can be implemented as a separate process or thread. Patterns in the second volume of the POSA series (*Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann [Wiley]) are useful for solving the communication issues inherent in separation into different threads or processes.

If the agents are separate processes, you can use patterns such as Proxies (see Chapter 19) to represent the agents and prevent dependencies. Or you can use the Forwarder-Receiver or Client-Dispatcher-Server pattern (see Chapter 21) to implement communication among processes. Interprocess control is inherently inefficient; you can mitigate this inefficiency somewhat by grouping subhierarchies of your application within process boundaries.

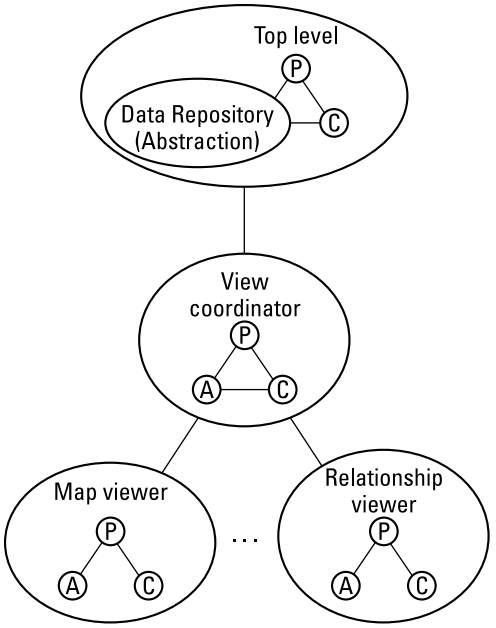


Figure 14-9: The coyote data system's rough hierarchy.

Step 3: Identify the top-level agent

In this step, you identify the functional core of the system. In the coyote research system, that core is the data repository, where all the data that the different displays need to use is stored. In other systems, the core may not be the data repository, which may be needed by only a subset of the subhierarchies.

The top-level agent contains two presentation-related capabilities: It's responsible for the top-level display selection and invocation, and it contains the UI elements that will be used across the whole system, such as common menu bars and dialog boxes.

Step 4: Find the bottom-level PAC agents

In this step, you need to go to the opposite extreme and identify the smallest self-contained components that the user can control or display. These agents are at the lowest level in the PAC agent hierarchy. In the coyote example, these components provide the different displays and provide the data entry spreadsheet.

Next, for each of these agents, identify the actual UI components to be used, such as menus, bar charts, and dialog boxes. These components will become bottom-level PAC agents.

Step 5: Find any bottom-level agents that aren't displays

Many systems have bottom-level agents that don't involve the UI directly. Instead, these lowest-level agents provide system services. The services aren't ones that all the other elements access regularly, such as the data repository; they're services that aren't directly related to the central focus of the system. In the coyote application, the error handler is the lone example of an extra system service agent.

Step 6: Compose lower-level agents with intermediate-level agents

In many systems (although not in the coyote research application), the concepts in the lowest-level displays are combined into larger sets that the user can operate on together in the UI. In this step, you create the intermediate-level agents that compose lower-level displays into these composite structures. These composition-focused intermediate agents allow the user to interact with the system in larger ways.

The coyote research example doesn't have any intermediate-level agents, because you're combining independent programs into a system. An example that *does* use an intermediate-level agent to combine bottom-level agents is an architectural drafting system. This type of system has bottom-level agents to display basic rooms. A house or other structure includes several rooms, and an intermediate agent allows the user to manipulate a group of rooms as a unit.

Step 7: Coordinate the lower-level agents with intermediate-level agents

Some system concepts have interrelated displays. Many text-editing programs, for example, have views such as page layout, outline, and web. If the user changes the underlying data while working in one view, the changes should be reflected in the other views as well. An intermediate-level PAC agent coordinates the information flow among the views.

Each of these intermediate agents may have its own UI, such as menu items. In the coyote research system, the view coordinator agent is one of these coordination intermediate-level agents. It uses the View-Handler pattern (see Chapter 20).

Agents also coordinate things other than displays. A system may start multiple concurrent jobs that an intermediate agent will coordinate.

Step 8: Separate the human computer interaction from the core functionality

Within each PAC component in this step, you separate subcomponents to provide the presentation and abstract capabilities. The presentation subcomponent is responsible for all parts of the agent's UI, including all the menus,

windows, and dialog boxes. The abstraction subcomponent stores all the data that is local to the agent and performs any calculations that are done on the data locally.

In some systems, the lower-level abstraction subcomponents use data provided by other PAC agents. In these cases, you can choose not to create an abstraction subcomponent or to create one with only basic functionality that accesses the data from the other agent. In the first case, you avoid replicating data across the system, and you avoid the effort of implementing the subcomponent. The second method incurs communication penalties, though: The PAC agents must share data, especially when actions like screen refreshes are performed.

After creating the presentation and abstraction components, you add the control subcomponent to mediate between them. The control is in place to prevent dependencies between presentation and abstraction subcomponents by providing an interface between them and adapting the data between them. The control component is best implemented as an Adapter.

In this step, your focus on the control component should be to manage the internal interactions within the PAC agent. In the next step, you add functionality to the control subcomponent to allow the agent to talk to other agents.

Step 9: Create the agent's external interfaces

Every PAC agent needs to communicate and coordinate with other PAC agents in the system, so in this step, you add this externally facing capability to the control subcomponent.

Inside the PAC agent, the control subcomponent must have some way to pass information to the other subcomponents. It also may have to pass information to higher- or lower-level agents.

A great way to implement this functionality is to use the Mediator pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*), because the role of the control subcomponent is to act as a mediator between the other agents and subcomponents.

In this case, you want a Mediator (from *Design Patterns*) rather than a Broker (see Chapter 12), because the Mediator will be connecting objects, not whole systems.

You can program the agents to exchange information in two ways. Weigh the complexity of each of the following methods against its benefits:

- **Create public interfaces to every service that an agent provides.** This method greatly simplifies the mediation role, because the external agents know directly what function to invoke and how to send or receive information.

  This method has a drawback, though: The interface of an agent can explode! An intermediate-level agent needs interfaces for everything it controls directly, as well as interfaces for everything that the higher- and lower-level agents provide, so although the direct interfaces makes things easier, the explosion of pass-through capabilities makes them more difficult.

  Another drawback of this method is that it introduces dependencies between agents. Each agent knows about the interfaces of the other agents. Therefore, changes to an agent can ripple through the other agents, making the system harder to maintain.

- **Implement a message-passing system.** This solution also becomes complex quickly. The control subcomponent needs to examine each message and decide how to process it: send it to the presentation or abstraction subcomponents within the same agent, send it to another agent higher or lower in the hierarchy of PAC agents, or process it within the control subcomponent. Deciding how to handle each message is complex.

  This method keeps the agent interfaces small — just the message-passing interface — so it reduces overall system complexity at the cost of increasing it within an agent. This solution prevents potential dependencies between agents, making the agents more reusable and easier to maintain.

The PAC agents act independently. Creating a registration process is an effective way to let the agents know about one another. For implementation details, see the Publisher-Subscriber pattern in Chapter 21.

Another aspect of cooperation between the agents is the responsibility of the control subcomponent: the notification of other agents when data has changed within an agent. This change-propagation mechanism also can use the Publisher-Subscriber pattern (see Chapter 21). Agents register with the agents containing data. When the data changes, the registration list is used for the agent to push — or publish — the data to the other interested agents that have registered to be notified.

There are other ways to share the changes besides the Publisher-Subscriber mechanism. The messaging and interface of the system can include information to propagate the changed information to the other agents in the system, for example.

Step 10: Tie the hierarchy together

By this point, you've structured your agents and built several subhierarchies. In this step, you tie everything together into the complete system by implementing the linkages among agents.

You must connect each PAC agent to all those lower-level agents that it cooperates with directly. If agents can be created or deleted dynamically, the intermediate-level agents that coordinate that creation and deletion must be given the functionality they need to do that work.

Chapter 15

Putting Key Functions in a Microkernel

In This Chapter

- Designing a simple, compact system core
- Implementing a microkernel

In this chapter, I tell you about a sophisticated way of building your system around a small core of functionality. This core, or *microkernel,* lets you build the outside layers from efficient, small, easily changeable parts.

Although operating systems (OSes) are the most common examples of microkernels, the technique is used in many virtual-machine infrastructure packages and is useful for solving a wide variety of other problems. Security appliances that screen messages with pluggable external servers to provide new security policies are another example. A database engine example has a microkernel providing core functionality to interact with the hardware and storage. Different conceptual views of the database are provided through policies implemented in external servers.

Problem: Hosting Multiple Applications

You've been assigned the task of designing an OS for a custom hardware device that your company is making. The OS won't be running general-purpose applications like web browsers or spreadsheet programs, so there's no need to port a commercial OS to the hardware. Also, because your company's hardware is custom, OS vendors aren't likely to provide a version for it.

The new system must be designed to integrate your company's applications into faster custom hardware. The applications already run in a Linux environment. (For this example, the current OS really doesn't matter; the problem is about the new system.) Linux provides the standard capabilities of OSes, such as scheduling, a file system, memory management, networking, paging,

and device drivers. The applications expect the existence of a file system, a process infrastructure, and management of the underlying hardware, but not much else. What you build must provide those capabilities to the new system.

Considering an existing OS

Your assignment is general enough that you can provide an environment for the applications on top of a commercial OS, or it can be something that you create in-house. It must provide the operating environment that your applications need.

The first thing you think about doing is porting a free OS to the new hardware. You could strip down a Linux distribution to get to the minimal functionality, because the applications put only minimal demands on the system. After you spend some time studying the problem, however, you think that the resulting system will still be bigger and more fully functioned than necessary. The central core, or kernel, of any widely available OS continually receives new features and capabilities. These updates increase the size of the OS and mean that you'll be removing unneeded features and fixes to unused parts of the OS all the time.

The other OSes that would be candidates for your system are commercial products. These candidates, however, have their own problems:

- You can't get source code for them, and without source code, you won't be able to port those OSes to your hardware.
- The hardware department's road map shows continual cost reductions to the hardware. This first release will use some components for basic functions that will be replaced by the third or fourth revision. Rapid evolution is another reason why the commercial OS solution doesn't seem ideal for your situation.
- As when you considered a Linux port, you find that a commercial OS doesn't have enough device drivers for your custom hardware, which would require you to continually adapt commercial (or third-party open source) device drivers to fit your environment.

These considerations lead you to decide to design your own OS.

Designing a custom OS

The first task in designing your own OS is thinking about what it needs to do to fulfill the applications' needs. What you build has to be portable,

extensible, adaptable, and small to allow easy evolution to future hardware (and software) technologies.

Because your hardware is a custom project built in-house, the hardware designers build only what's needed, not putting in any extra frills. You know that this simplicity means that the applications and your OS should have small memory footprints and be efficient in terms of processing.

Separating policy from mechanisms

Currently, all the applications that you want to support in the new OS are built on top of a full-function OS. You can imagine that in the future, the system will support several other applications that your company currently runs on different OSes. So, over time, your custom OS needs to run standard applications that may expect different operating environments, so it needs to emulate several existing standards.

Another criterion for the system you're building is that you make it easy to add applications. You like the ease of plug-and-play hardware and software plug-ins, for example, and you want it to be that easy for an application to move over to your new OS.

You're going to build the mechanisms to provide services to your OS. These services will be *atomic* — that is, contained and bounded. To make your OS usable by applications, you must ensure that the core of the OS provides the basic mechanisms that the applications used in their previous, full-function OSes, because the applications expect an OS to implement basic functions in a certain way.

On top of these basic mechanisms, other parts of your design need to provide higher-level capabilities that build on the mechanisms to implement a policy.

To support the applications that currently run on several OSes, you build a system like the one shown in Figure 15-1. You implement the mechanisms that are common to all the OS policy levels. You also build the policy level for the needed OSes. The policy level interfaces between what your current applications expect of an OS and the actual mechanisms that your small OS provides. Because you have applications coming from several commercial host OSes, you decide to build the underlying OS as a modular core of mechanisms and also build the policy layers for two widely used OSes: Windows and Linux.

The policy layer encapsulates the functions that your applications expect the OS to provide. It translates requests from the applications into calls to the mechanisms that your OS supplies.

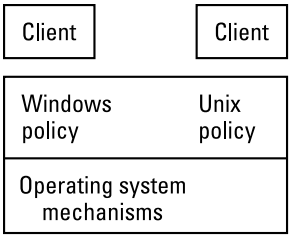


Figure 15-1: OS and policy layers.

The policy layer and the OS layer provide application programming interfaces (APIs) to their policies and mechanisms, respectively. At the policy level, these APIs mimic a subset of the commercial OS; at the OS level, they provide the basic functions required by all the policy layers.

Building the system

Because you want to keep the core of the OS simple and small, it should provide only the most necessary mechanisms that are common to all users of the system. Other functionality will be supported through servers.

Core

Basic hardware interfaces are provided in your core OS. Some hardware associated with the system still isn't common to all the operating environments, however, so you build interfaces to this hardware separately from the OS. Only one of the applications being ported to the new hardware needs Bluetooth technology, for example, so it doesn't make sense to put Bluetooth support in the core of the OS.

Servers

Instead of adding more functions to the core, you keep the OS to a minimum size and introduce servers — either internal or external. These servers add other functionality without increasing the size of the core OS (see Figure 15-2). The policy layer is a specific example of a server that you build.

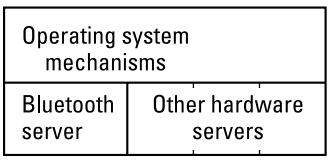


Figure 15-2: The core OS with internal and external servers.

External servers are separate processes. These servers look to the outside world like part of the OS because they provide functionality that the applications expect to be part of the OS, at least with respect to the policies for which that particular server is responsible. The policy layer is an example of an external server.

The external servers provide interfaces that the client applications want to execute, such as APIs for the file system. The actual file system is implemented as an internal server. It's internal because the clients don't talk to it directly — they talk to the core, which then talks to the internal server. The file system is a server because it provides common functionality that isn't present in the core of the OS.

Internal servers are like external servers in that they interface between the OS and some other functionality rather than between the client and some functionality in the core OS.

Clients

The clients of the system are the applications that use the system. The clients communicate with your OS by using communication facilities provided by the core of the OS, and your OS communicates with the servers that provide functionality not available in its core. Figure 15-3 illustrates this communication hierarchy.

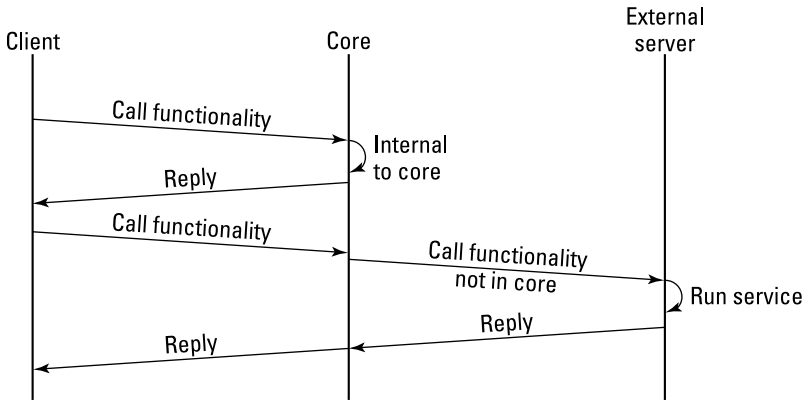


Figure 15-3: A client talks to the OS core, which talks to servers.

Adapters

You also add adapters to the system. These adapters slide between the clients and the core of the OS, as shown in Figure 15-4. The goal of adapters

(sometimes called *emulators*) is to shield the user of the system from the internal details and, hence, from depending on certain implementations. Adapters run in client address space and adapt the application to the programming interface of the external server.

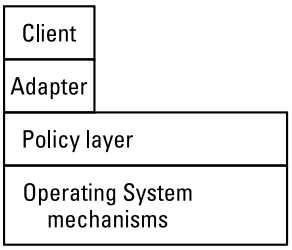


Figure 15-4: An adapter between OS and client.

Extensions

With this arrangement of clients, servers, and the core OS, adding extensions is easy. The places to add new functionality are well defined:

- **Core or microkernel of the OS,** if the extensions are indeed core mechanisms that all the servers need
- **Internal servers,** if the functions aren't going to be externally visible to the clients
- **External servers,** if the functions are internal to the system but visible to the clients
- **Adapters,** if the extension adds something new between the application and external servers
- **Clients,** if the functionality is related strictly to a single application

Solution: Building Essential Functionality in a Microkernel

Microkernel architectures adapt to changing system requirements easily. A microkernel core with a small set of functions is enhanced by extensions that give the system more and customer-specific functionality.

The microkernel is the core OS that I've been telling you about in this chapter.

Examining Microkernel Architecture

The section of this chapter that describes the problems associated with creating a custom OS introduces the parts of the microkernel. In this section, you dig deeper into the parts of a microkernel-based system.

Viewing the architecture's parts

To implement microkernel architecture, you need to design five kinds of participating components:

- Microkernel
- Internal servers
- External servers
- Clients
- Adapters

I describe these components in detail in the following sections.

Microkernel

The microkernel is the main component of this architecture, providing the core of basic, essential services. The microkernel encapsulates hardware-specific parts, shielding the applications, clients, and servers from these parts. It manages essential resources such as processes or files, providing access to those resources through atomic services, or *mechanisms.* These mechanisms form the basis for building higher-level *policies,* which provide more complex functionality.

Figure 15-5 shows the Class-Responsibility-Collaboration (CRC; see Chapter 2) card for a microkernel component.

Class Microkernel	***Collaborators*** • Internal server
Responsibilities • Provide core mechanisms • Provide communication facilities • Encapsulate system dependencies • Manage and control resources	

Figure 15-5: Microkernel CRC card.

Internal servers

Internal servers are components that extend the functions provided by the microkernel. These servers are closely related to the microkernel and may be tied to hardware and underlying platform capabilities. The microkernel invokes the functionality provided by the internal servers through service requests. The internal servers encapsulate the capabilities of the underlying system and help keep the microkernel small. Device drivers are examples of internal servers.

Keeping the microkernel small in terms of memory and fast in terms of processing time is one of the goals of this architectural style. This goal is the reason why the internal servers are created: to isolate additional, complex, and possibly optional services from the small, fast core.

Internal servers are extensions of the microkernel and are accessible only through the microkernel; they aren't accessible to clients directly.

External servers

Unlike the internal servers, which must be accessed through the microkernel, the external servers export interfaces, which allows clients to invoke them directly. These servers implement the policies that are built on the basic mechanisms provided by the microkernel itself. External servers usually run as separate processes; they access the microkernel API to accomplish tasks for the client.

Collections of external servers that work together to implement their own view of the underlying application domain are sometimes called a *personality*. They provide a layer of abstraction on top of the microkernel, offering different policies through different implementations.

Figure 15-6 shows CRC cards for both internal and external servers.

Class Internal Server	***Collaborators***
Responsibilities • Implement additional services • Encapsulate system specifics	• Microkernel

Class Internal Server	***Collaborators***
Responsibilities • Provide programming interfaces to clients	• Microkernel

Figure 15-6: Internal and external server CRC cards.

Clients

Clients are the applications using the system. They communicate with the microkernel-based system only through the external servers.

There's a chance that clients will become too closely tied to the external servers. One way to prevent this from happening is to introduce adapters into the system (see the next section).

Adapters

Adapters provide an interface between clients and external servers, reducing the risk of tight coupling. Too-close coupling makes it harder to change the system without also changing the client application's functionality.

The adapters run in client address space, rather than in server or microkernel address space, so you should think of them as being part of the client. They allow the client to integrate with the given external server without any modifications — that is, they shield the client from changes. The adapter receives requests from a client and passes them to the appropriate external server.

Plug in a game

Games can be purchased on cartridges or special cards for a number of gaming consoles, like Nintendo's Game Boy, DS, and 3DS consoles or the PlayStation Vita. The core functionality is built into the actual gaming console that provides the computing capabilities, the displays, speakers, microphones, cameras, and other input/output capabilities. The cartridges or cards provide the specific games and sometimes the ability to store game status and high scores.

The parts of the game console and game card system map onto the Microkernel pattern like this:

- **Player:** The player who uses the combined game and card is the client of the Microkernel system.
- **Gaming console:** The console itself (without the game card) corresponds to the microkernel.
- **Game cartridges or cards:** The cartridges or cards are internal servers that provide extra functionality to the microkernel (the gaming console).
- **Pin connectors between console and cartridge:** The actual pins that make the electrical connection between the console and game cartridge are adapters in the microkernel system.
- **Game console buttons and input devices:** Because the buttons and touchpads of the game console provide an interface for the client (player), they correspond to external servers.

This architecture of games (and microkernels) allows the functionality to be changed easily through the changing of servers, without changing the heart of the system.

Figure 15-7 shows CRC cards for clients and adapters.

Class Client	***Collaborators***
Responsibilities • Provide user application	• Adapter

Class Adapter	***Collaborators***
Responsibilities • Hide system dependencies from the client • Invoke external server methods on behalf of clients	• External server • Microkernel

Figure 15-7: Client and adapter CRC cards.

Exploring the effects of the Microkernel pattern

Microkernel architectures are very good at what they do: providing a small, modular framework for an OS or a similar environment. In this section, you see the benefits of microkernels as well as the liabilities that go along with using the microkernel approach in your architecture.

Benefits

These benefits are visible if you use a microkernel:

- **Microkernel architectures are portable.** To move the microkernel to a new hardware or software platform, you need to change only a small part of the microkernel, because the small functional core has few hardware dependencies and selected servers where most hardware dependencies are confined. Also, a clear boundary exists between the microkernel and any servers and applications. All the platform dependencies are in the servers, not in the clients and applications.

  You may need to rewrite the actual microkernel code and some internal servers extensively as part of a port — but they're only a small part of the overall application. The external servers, adapters, and clients shouldn't require much revision for porting.

- **Microkernels are extremely flexible.** You can extend them easily by adding new servers to provide new functionality. All OSes are designed to make running applications easy, but with microkernels, the OS is so small that it's easy to extend.

- **Separating policy from the mechanisms needed to implement policy is easy with a microkernel.** The microkernel contains only a small set of core functions, which provide services to the servers and clients to achieve the real results. The small set of core functions is enough to satisfy the needs of the clients, which then can implement whatever they want, without the limitations that would apply if the microkernel enforced its own policy. (For more information, see "Separating policy from mechanisms," earlier in this chapter.)
- **Security and reliability are enhanced.** Normally, the microkernel is run in protected address space, and everything else — all the servers, adapters, clients, and applications — runs in separate process spaces. This arrangement keeps the parts isolated from one another and prevents inappropriate interactions.

The distributed-microkernel variant (see the sidebar "Considering microkernel variants," later in this chapter) provides even more benefits:

- **The same capabilities that make microkernel-based systems easy to extend in terms of functionality make adding new instances of the microkernel itself easy as well.** As a result, scaling the system is easy.
- **Reliability is enhanced.** The distributed-microkernel architecture makes it easy to replicate the system. Having more microkernels supporting more instances of an application also enhances the availability of the application.
- **Transparency is enhanced.** In a distributed system, the microkernel and the adapters don't have a lot of extra, unnecessary functionality.

Liabilities

No pattern is without some liabilities that must be balanced with the benefits. Here are the liabilities of using a microkernel:

- **A monolithic system generally has higher performance than a microkernel system does.** You pay a price in performance for the flexibility and extensibility of a microkernel. Microkernels that are adapted to specific hardware, however, make high performance possible.
- **A microkernel is complex.** Designing a microkernel isn't a trivial process; both analysis and building the set of core functions can be difficult. You need in-depth knowledge of the system and applications during analysis and design.

Considering microkernel variants

Two variants of the microkernel architecture are commonly used:

- **Message backbone:** In the message-backbone variant, the connections between client and server are indirect. All requests between a client and a server pass through the microkernel, which acts as a communication pathway.

  This variant is especially useful if your environment requires all messages to go through a central hub. This may be the case for security (so that communications can be screened) or regulatory compliance (so that all the messages can be logged).

- **Distributed microkernel:** In the distributed-microkernel variant, an entire large system appears to the user to be a single microkernel system, although in fact, it's an assembly of components, each of which has its own microkernel implementations. Messages are exchanged among the microkernels via the message backbone (see the preceding item). To build this kind of system, you must give the microkernels communication-related services that they may not have otherwise.

Implementing a microkernel architecture

In this section, I explain the 12 steps required to implement a microkernel architecture.

Step 1: Analyze the domain

The first thing you need to do as you implement a microkernel architecture is understand the domain. Perform domain analysis (see the nearby sidebar) to understand the core services that clients in the domain expect, such as support for specific devices or specific standard suites of APIs.

Here are some questions to ask and answer:

- What common OS-level functions do all applications expect? (The answers may include file systems, memory management, virtual memory management, and paging.)
- What are the features that you expect to be required as common functionality — but that you realize don't *need* to be common?

Step 2: Categorize the services

In this step, you look at all the services that the microkernel must supply and create separate categories for groups of those services.

Domain analysis

Domain analysis is the analysis of a related set of solutions or systems in a product line to identify the commonalities and variabilities. The *commonalities* are those things that are the same across all the examples in the set of systems. The *variabilities* are those things that change between the examples.

Domain analysis is useful because the commonalities point to functionalities and capabilities that can be developed in common and shared across a family of products. The components that implement the common capabilities are ideal candidates to be reused by all the products in the family. The variabilities are emphasized or deemphasized in products to differentiate the products from one another.

Some categories are those needed by the applications, which are candidates for migration to external servers. Another category is those needed for the microkernel infrastructure, which are candidates for migration out of the core to internal servers. Yet other categories are the fundamental items, which you should include in the microkernel.

Step 3: Partition the categories

In this step, you refine your category list by sorting the categories into those that will be implemented by the microkernel and those that will be implemented by internal servers.

Don't divide the categories arbitrarily; decide on specific criteria.

Here are some example criteria that you can use:

- Small, fast microkernel
- Time-critical components in the microkernel
- Frequently used functionality in the microkernel
- Hardware-dependent functionality categories to internal servers

Step 4: Identify the microkernel's mechanisms

Review the domain analysis from Step 1. Every function that the application expects must become a policy of an external server. Continuing into the core of the system, every mechanism that the external servers need to implement the policies must be provided by the microkernel or as a policy of some other server.

In this step, identify the mechanisms that the external servers need from the microkernel to implement their policies. What you're doing in this step is defining the interface to the microkernel.

Step 5: Define communication strategies

Now you must decide how the microkernel will provide communication among the parts of the system. Communications can be asynchronous or synchronous, one-to-one, many-to-one, or many-to-many, depending on the applications' needs.

In many cases, the building blocks that you use to create your microkernel supply a certain messaging paradigm, such as message-passing or shared memory communications, that you can build on. See the Forwarder-Receiver and Client-Dispatcher-Server patterns in Chapter 21 for information about how to implement two styles of communications infrastructures.

Step 6: Structure the actual microkernel component

Use the Layers pattern (see Chapter 9) to separate the system-specific parts from the system-independent parts of the microkernel. Put the services that the other components of the system use in the topmost layer, and hide system details in the lower layers, as shown in Figure 15-8. The details that you put in lower layers are those that are more likely to depend on the particular system or hardware.

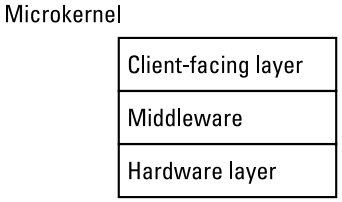


Figure 15-8: A layered microkernel.

Step 7: Define the microkernel's programming interfaces

How and what should be accessible to external servers? This step is the time to decide.

Base your decision on the technology that you're using to implement the microkernel: a separate process or a shared process. If you're using a shared module, you can use ordinary method calls for communication. If the microkernel is a separate process, you must create the interprocess communication facilities needed for the microkernel to talk to the servers.

The microkernel can become a bottleneck in the system because it's an exclusive resource. There's only one microkernel, after all. You can mitigate this problem by providing multiple threads that wait for requests and other threads that execute services. You must make sure that your implementation is thread-safe and that resource integrity is preserved.

Step 8: Manage system resources

All system resources are handled with unique identifiers from the adapters, clients, and/or servers by the microkernel. The microkernel maps from the identifiers to the resources.

Step 9: Design and implement the internal servers

To design the internal servers, you can use either separate processes or shared libraries. Create the internal servers in parallel with steps 7 and 8 because of the close relationship between the microkernel and the internal servers.

Only the microkernel can communicate with internal servers.

Your internal servers can be either of two kinds:

- **Active:** Active servers are implemented as processes, so design them as event loops. If the server receives a request, it interprets that request, executes it, and then resumes looping.
- **Passive:** Passive servers are implemented as libraries, so call them by invoking their interface.

Step 10: Design and implement the external servers

External servers receive requests, analyze them, execute the requested services, and send results back to the client. They may call mechanisms in the microkernel. Typically, an external server is implemented as a separate process with its own service interface.

You need to define how requests are dispatched to internal procedures. One way is to integrate a dispatcher with the main loop that unpacks events and then calls appropriate procedures via a callback. See the Reactor pattern in *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley), for more information about implementing this event-driven approach.

Step 11: Implement the adapters

Adapters provide functionality to the clients by forwarding that functionality to an external server. The clients are calling the servers. Behind the scenes,

an adapter packages any extra information that a server needs and sends it to the appropriate server.

Adapters can be statically or dynamically linked libraries. The adapters interface with exactly one external server (see the Proxy pattern in Chapter 19).

One way to optimize the system is to allow adapters to communicate with the microkernel on their own for some operations instead of forwarding all requests to the external server. Another way to optimize is to cache the responses to common requests.

Another trade-off that you need to consider in this step is whether one adapter should represent all clients or only one client. One adapter for all clients is better in terms of memory but requires extra processing to forward messages to the correct client. If every client has its own adapter, more memory will be used, and the system will have more components, but response time will be better because the dedicated relationship simplifies the communication.

Step 12: Develop client applications

The last step is designing the applications that will use the microkernel architecture. Sometimes, you can reuse existing applications. The applications invoke clients that interact with the system through the policies offered by the external servers.

Chapter 16

Reflecting and Adapting

In This Chapter

- Getting acquainted with reflection
- Finding reflection in the real world
- Designing and implementing the Reflection pattern
- Using the Reflection pattern in modern programming languages

In this chapter, I tell you about reflection and the Reflection architectural pattern. *Reflection* is the ability of a program to inspect its own internal structure and to modify its internal structure and behavior. It's a useful concept when a program needs to be very adaptable and easy to evolve to changing requirements. It's also useful when a program must analyze its own behavior for other reasons.

Many people are confused about the general technique of reflection. They don't understand what it is, how it's used, or how powerful it is. The first section explains the basics of reflection so that when I start talking about the architectural uses of reflection, you'll have the same understanding of the concept that I do.

Reflection, the architectural pattern that I discuss later in this chapter, is useful when you want your application to be able to change itself at runtime. This pattern opens the door to applications that adapt when you change your mind about what you want them to do.

When I want you to focus on the architectural pattern, I capitalize the word *Reflection;* when I'm discussing reflection in general, I leave the word lowercase.

Understanding Reflection

Reflection draws a distinction between the base level of the program that doesn't change and a meta level that does change. The *base level* implements the application logic and makes use of information from the meta level. The

meta level encapsulates the internal parts of the application that can change in *meta objects.* The meta level's data (the *metadata*) describes an application's attributes and behavior that can change.

The base level of the application interprets the metadata at runtime to adapt the application and include its new structural behavior. To get a complete view of the application, you must look at both the base-level objects and the current configuration of the meta objects.

Figure 16-1 shows two display instances created with identical base-level objects and different metadata. In this figure, the metadata is expressed with Extensible Markup Language (XML), but XML isn't the only way to represent metadata. The metadata describes the shape of objects by using only points and connections between points. The base level understands, and uses, two structural aspects of the application to create the desired objects. There are no specific predefined classes that take precisely three or four points; the base-level objects create new classes with as many points and lines as defined in the metadata. Arguably, this simplistic example could be done without reflection; what makes it reflection is that the internal representations of the objects are new classes that haven't been preprogrammed.

Reflection has another level in addition to the base and meta levels: the *meta-object protocol* (MOP), which defines how changes to the meta level are made. In Figure 16-1, the MOP is handled in an XML editor that isn't shown.

Reflecting on the Constitution

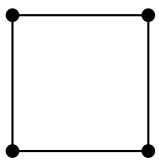
The Constitution defines the rules that govern the behavior of the Congress. To change the rules for Congress, the Constitution must change. Changing the behavior of a system from within the structure of the system is *reflection.* The Constitution also defines how to change the Constitution by adding amendments, which makes it possible to change the rules of behavior of Congress. This change doesn't change the people who make up the Congress or the location where Congress meets.

The Constitution corresponds to the meta object. The actual people and the place that Congress meets correspond to the base level. Changing the base level's behavior involves changing the meta object and doesn't involve changing the base level. The meta-object protocol is the rules for changing the Constitution (the meta object).

Changes to the system are easy to make through the meta-object protocol. Many kinds of changes are possible without explicitly changing the actual base level. Changing the meta object can be dangerous, though, because the changes can damage the system — just as changes to the Constitution can cause undesired consequences.

Meata data example 1

```
<object>
  <name> SQUARE </name>
  <point>
    <id> 1 </id>
    <x>  0 </x>
    <y>  1 </y>
  </point>
  <point>
    <id> 2 </id>
    <x>  1 </x>
    <y>  1 </y>
  </point>
  <point>
    <id> 3 </id>
    <x>  1 </x>
    <y>  0 </y>
  </point>
  <point>
    <id> 4 </id>
    <x>  0 </x>
    <y>  0 </y>
  </point>
  <connection>
    <connection_id> 1 </connection_id>
    <from> 1 </from>
    <to>   2 </to>
  </connection>
  <connection>
    <connection_id> 2 </connection_id>
    <from> 2 </from>
    <to>   3 </to>
  </connection>
  <connection>
    <connection_id> 3 </connection_id>
    <from> 3 </from>
    <to>   4 </to>
  </connection>
  <connection>
    <connection_id> 4 </connection_id>
    <from> 4 </from>
    <to>   1 </to>
  </connection>
</object>
```



Meata data example 2

```
<object>
  <name> TRIANGLE </name>
  <point>
    <id> 1 </id>
    <x>  0 </x>
    <y>  0 </y>
  </point>
  <point>
    <id> 2 </id>
    <x>  0.5 </x>
    <y>  1 </y>
  </point>
  <point>
    <id> 3 </id>
    <x>  1 </x>
    <y>  0 </y>
  </point>
  <connection>
    <connection_id> 1 </connection_id>
    <from> 1 </from>
    <to>   2 </to>
  </connection>
  <connection>
    <connection_id> 2 </connection_id>
    <from> 2 </from>
    <to>   3 </to>
  </connection>
  <connection>
    <connection_id> 3 </connection_id>
    <from> 3 </from>
    <to>   1 </to>
  </connection>
</object>
```

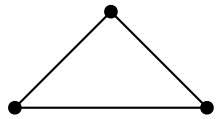


Figure 16-1: Two different applications created through metadata.

In this chapter, Reflection is something that you use to design your overall application. Many programming languages implement capabilities that they call reflection. The idea is the same — these capabilities examine a program and change its structure and behavior — but not every use of a programming language's reflection capability is an architectural design choice. The ways to use nonarchitectural reflection effectively within a language are features of a language, or *idioms*.

Looking for Reflection

To help you understand reflection, this section provides several examples. Table 16-1 breaks these example applications into the three parts of reflection: base level, meta level, and MOP.

Table 16-1 Reflection Examples

Example	*Base Level*	*Meta Level*	*MOP*
Externalization	Rich-typed application; raw typeless input and output	Type conversion code	Runtime type information; object system itself
Code analysis tools	Code being analyzed	Classes that count and check code being analyzed	User interface (UI) for analysis tool; UI for information retrieval
Aspect-oriented programming (AOP)	Code being developed	Handlers and AOP infrastructure	Protocol for adding handlers and AOP choice points
System configuration files	Static parts of system	Variable settings and parameters	Configuration file and editor

For details on the use of the base level, meta level, and MOP as classes in the Reflection pattern, see "Designing Architectural Reflection," later in this chapter.

Externalization

Most computer programming languages allow you to create groupings of related data in the form of types, structures, or classes. Within a program,

these structures are great — they implement several of the enabling techniques that I discuss in Chapter 2, such as encapsulation, information hiding, and modularization.

The problem comes when you need to store that structure in a file or send the data structure over a serial communication link. In these cases, the nice internal structure that your program uses must be *externalized* — converted to a serial stream of bits (or converted from a stream of bits to the structure again).

I discuss this problem in conjunction with several other architectural patterns in Part III of this book. Look for terms like *serialize* and *marshal.*

Today, many libraries, modules, and open-source software applications are available to do the serializing and deserializing of the data, such as `marshal` and `yaml` in Ruby, and `hibernate` or the `JSON` libraries for many other languages. These serialization and deserialization components don't need any previous knowledge of the structures; the programs use reflection to explore the structure by looking at the structure rather than preprogramming. You can use the built-in reflection capabilities of many programming languages to build your own serialization and deserialization capabilities.

Some languages, such as C++, don't support reflection natively, but it's still possible to use reflection for externalization in those languages. For an example of using reflection in C++ to provide flexible externalization, see the chapter on the Reflection pattern in *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley).

Code analysis tools

Code analysis tools use reflection to improve their understanding of the program being analyzed. When an analysis tool is built into the application that's being analyzed, reflection improves the tool's ability to analyze the software.

A simple example of reflective capability in an analysis tool is the capability to dump the contents of an unknown object. (This example is similar to the earlier example in which an unknown object must be externalized.) Reflection also allows the code analysis tools to look at code without predefining all the types. Finally, through reflection, the tools can add hooks or code to watch certain events.

An example application that uses reflection is a student programming assignment analyzer (see the nearby sidebar).

Analyzing student programming assignments

An interesting application of reflection in the Java language is a Java student program analyzer. The goal of the analyzer isn't to grade the programs automatically; instead, the goal is to lead the students to make consistent formulations of the problem, which ultimately makes manual grading easier. It encourages and supports the students to use the same names for required elements of the programming assignment. A side benefit is that it gives the students reinforcement along the way, assuring them that their work is progressing down the correct path. The analyzer enforces similarities among the individual student programs so that the instructor doesn't have to spend a large amount of time understanding each program's basic structure. Instead, she can spend her time grading the assignment, not learning each student's naming conventions.

After giving an assignment with specific requirements for naming or structuring some parts of the solution's implementation, the instructor creates a reference implementation. This reference implementation complies with the assignment requirements and includes some attributes that the analyzer uses to extract and create JUnit test case classes — an example of reflection in the program. Then the instructor gives the test classes to the students. When the students run the JUnit test suite, the tests use reflection again to determine whether each student assignment program matches the specified attributes — assignment requirements — of the reference implementation.

For more information about the analyzer, visit `www.twodee.org/speccheck`. To find out more about the open-source JUnit project, see `www.junit.org`.

Aspect-oriented programming

Aspect-oriented programming (AOP) is a way of handling characteristics of a program that are separate from the main purposes of the program. Capabilities such as debugging, fault tolerance, and logging require little bits of functionality to be placed in many places across the main application, and AOP can perform that task.

To use AOP, you write a specification that includes locations in the existing program where these separate handlers and checks should be inserted. The idea behind AOP is that some aspects of an application are hard to write into a well-structured program because they cut across the main structure. Reflection can help the AOP language automatically add hooks and code in the right places in the application to achieve functionality that cuts across the main application.

System configuration files

Reflection can be useful for dynamically adding or creating program elements based on configuration files. When you're building an application or framework that must work seamlessly with code that's not yet written, the application can use reflection to examine the new code it must work with and modify itself to interact correctly with the new code.

On a more basic level, your system can have configuration files that define how the system should behave. This behavior exists in some versions of the Windows Registry. The Registry is part of the system that other parts examine to configure their environments, and the Registry is updated to change future behavior as well.

Many development tools, such as Eclipse, use reflection to build up the configuration that you need.

Designing Architectural Reflection

An application built of software is static and does the same thing every time it executes. Sometimes, however, you want to be able to reconfigure the application easily and to make it adaptable. Here's what I mean by an *adaptable* application:

- An application to which you can add new capabilities without rewriting or extensively changing the whole application
- An application that can evolve to meet new technologies and new customer needs so that you can incorporate the latest ideas and customer requirements into the application easily, without throwing out the whole application and starting over

Making applications adaptable

Changing and evolving software is a tedious, error-prone endeavor. It's also expensive, because the developers who are handling the evolution must spend lots of time understanding the software's current behavior before they can change it.

The internal structure of applications that adapt easily to change is very complex. Maintaining these applications is difficult because of the many components used to encapsulate changes. Providing additional ways to perform

adaptations and growth also increases complexity. Modern programming languages provide many built-in ways to adapt and grow software, such as mix-ins, subclasses, templates, and parameterization. You also resort to good old-fashioned cut-and-paste reuse and adaptation — in other words, copy parts of code from one place and alter it for its new functionality.

Adapting an application can require making anything from small changes to massive revisions. The modifications can involve changing one object, or they may require touching everything in the entire application to make the change. Modification and adaptation can even touch the core of an application, such as the communications infrastructure. This makes adaptation more difficult, especially if the program needs to adapt at runtime instead of at compile time. Changing out the base infrastructure is not for the faint of heart.

The way to solve this problem of adaptation, which is making the problem easier to solve and with a better structure, is to make the software self-aware. As you design the software, identify and target certain aspects to be changeable. Consider the changeable parts to be part of the meta level. The base level is the core application that isn't changed; it uses structures and data defined by the meta level.

Self-awareness can be either introspection or reflection:

- **Introspection** is a program's capability to examine itself dynamically.
- **Reflection** is a program's capability to examine itself and its own data and to make changes, including changing and adding to class definitions dynamically, as well as modifying its own behavior.

Structuring the classes

Three different types of classes are involved with Reflection:

- **Base level:** The base level is comprised of the parts of the application that perform the basic algorithms of the application — the parts that implement the application logic. The base level can be changed by the meta level but doesn't actively change itself. The base-level classes are independent of the changes that may occur at the meta level. Base-level classes access changeable information by interacting with meta objects. The base-level objects don't store the changeable information themselves.
- **Meta level:** The meta level encapsulates the parts of the application that may change to create new applications from the existing base-level classes. Classes at the meta level may cause changes to be made to

other meta-level classes, to base-level classes, or even to the way that base-level classes interact with meta-level classes. The state of all the meta-level classes together with the base-level classes describes an application.

- **Meta-object protocol (MOP):** The MOP abstracts the meta-level classes to make the changeable meta-level classes accessible externally. The existence of a MOP makes changing the meta objects possible. The MOP has access to the internal workings of the meta objects, which allows you to change meta objects and the way that base-level objects behave. The MOP also can change the connections between base-level and meta-level objects, but to do this, it must be able to modify the base-level objects, too.

  Table 16-1, earlier in this chapter, shows that in some cases, the MOP is an ordinary editor that changes configuration variables. In other cases, you must create a specialized interface to allow the MOP to adapt applications.

Class-Responsibility-Collaboration (CRC) cards for these three classes are shown in Figure 16-2. (CRC cards were introduced in Chapter 2.)

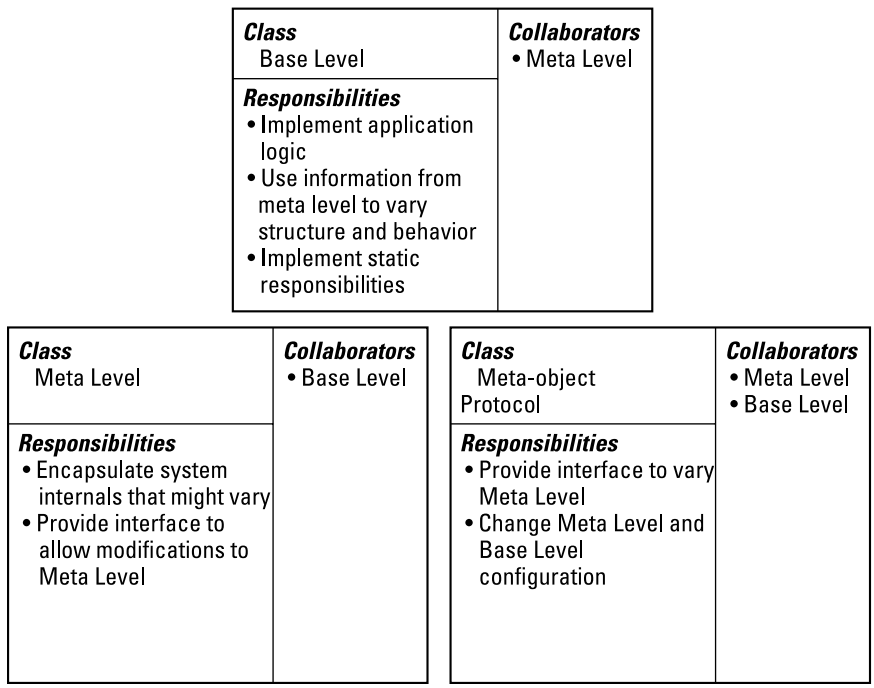


Figure 16-2: CRC cards for the three types of components in a reflexive application.

Understanding the consequences of Reflection

Reflection is useful in many circumstances. In "Looking for Reflection," earlier in this chapter, I list some examples. Reflection benefits these applications but also has some drawbacks, as you see in the following sections.

Reflection should be used in moderation. If you don't use it correctly, the application may become unstable and hard to maintain, because the people responsible for the application may not understand how it has changed itself.

Benefits of Reflection

Here are the general benefits that you'll find when you use architectural reflection:

- **You don't need to modify the software explicitly after you've created an adaptable application that uses Reflection.** You don't need to modify the existing code when you adapt the application, because you can make changes by calling a function in the MOP. The MOP is in charge of getting your changes into the meta objects and the revised meta objects into your application.
- **The MOP makes changing the application safe by providing a consistent interface to perform the adaptation.** It hides the complicated details of the adaptable application. A well-designed MOP helps prevent unsupported changes to the application.

  If you use a general-purpose MOP like an XML editor, you won't realize this benefit unless you also build a validity-checking tool.
- **Changes at almost every scale are possible in an application that is constructed around the Reflection pattern.** Reflection helps adapt the software to a changing environment and to changing customer requirements.

Drawbacks of Reflection

Like all patterns, the Reflection pattern has negative consequences. The following list explains things that you need to watch out for as you design and build a reflective application:

- **You can make damaging changes to the application through incorrect changes at the meta level.** This danger emphasizes the need to construct a good MOP. An example of an unsafe change is changing a database schema without first stopping the parts of the application that are accessing the database.

- **Applications built around reflection have more components.** In some cases (especially the applications I talk about in the sidebar "Working with adaptive object models," later in this chapter) the number of meta objects is greater than the number of objects in the base level. This situation isn't always bad, but maintenance becomes more difficult when you have more objects to maintain.
- **Reflection requires extra processing.** This extra processing is required for tasks such as retrieving configuration information, changing the meta objects, and ensuring the consistency of the application and internal base level with meta-level communications. Applications with reflection can have lower efficiency because the base-level objects must check with the meta objects regularly to retrieve configuration information that will adapt their behavior.
- **Reflection capabilities must be programmed into the application in the beginning.** Ongoing adaptation involves changing meta objects. Some capabilities that you might like to change today may not have been designed to change when the software was created; as a result, the application may not support your desired changes.
- **Reflection capabilities aren't supported in every programming language.** I discuss this drawback and what it means in "Programming Reflection Today," later in this chapter.

Implementing Reflection

I don't provide detailed implementation information in the following seven steps because so much of building an application with the Reflection pattern is language-specific. Instead, in this section, I provide a checklist of things to consider as you design your reflective architecture in any language. In the last section of this chapter, "Programming Reflection Today," I give you some examples and pointers on using reflection that will help you with the implementation steps in this section.

To illustrate these steps, I use the example of creating the part of a web-based sales application that displays varying product information. The types of items to be sold are described with XML-based metadata. The application looks at this metadata to determine the product information that should be displayed on the website. The metadata includes both product attributes and instructions for displaying those attributes on the web, using low-level characteristics known to the base-level classes.

You could provide this functionality without reflection, but if you did, you'd have to preprogram the application for every possible kind of product data in advance. By using the Reflection pattern, you can add new attributes later, during runtime, including descriptions, product features, and feature types and totally new fields that you never imagined when you built the application.

Iterate through Steps 5, 6, and 7. Especially when you're just starting to use Reflection, this iteration will help you design your application with effective structuring of your base level, meta level, and MOP.

Step 1: Define the application's model

Using an appropriate analysis method, start by understanding the problem fully. You should understand the following aspects of the application:

- **Services:** The services that the application is supposed to provide
- **Components:** The components that you need to design to support these services and the relationships among the components
- **Component cooperation:** How the components collaborate and cooperate with one another
- **Data:** The format of the data used by the components
- **User interaction:** The means by which the user will interact with the application — both at base level and meta level

The primary service of interest in the web-sales example is the display of the product description. (The actual web server is beyond the scope of this example, as are the communication methods.) You have a set of components that together interpret the product metadata and prepare the display. The data aspect of interest is the metadata design (see Step 5). Buyer interaction with the application occurs via the web interface. Management interaction — defining new products — is done through the MOP that you design in Step 6.

Step 2: Identify what behavior will adapt

In this step, you decide what aspects of the application can change. Examine the application model that you created in Step 1 to understand which application services may change and which must remain constant. For this analysis, you can use domain analysis (see Chapter 15).

There aren't any hard-and-fast rules about what can vary and what must remain constant; something that changes in one application may be the most stable part of a different application. The aspects that change are based on the application and the application's environment.

Here are some examples of the kinds of things that can change in applications (although not necessarily in your application):

- Real-time constraints such as deadlines, protocols, and real-time algorithms
- Protocols that define the behavior of transactions
- Interprocess communication mechanisms

- Exceptions, error handling, and fault-tolerance mechanisms
- Algorithms that change between instances of the application, such as to account for regional or national characteristics (maybe tax rates)

In the web-sales example, the display of the product data will change based on the product definitions stored in the metadata. (This example doesn't have as large a behavioral change as you'll design into some applications.)

Step 3: Identify the application's structural aspects

Identify the structural aspects that need to be defined and changed at the meta level without affecting the base level of the application. You design the base level to depend on meta objects to define theses aspects but for the base-level objects to function correctly no matter how the aspects change.

The basic framework of the web-sales product display is a structural aspect that shouldn't change. It provides a framework around all products that is guaranteed to be present.

Step 4: Find the varying system services

Find the system services that support the variations of the system you identified in Step 2 and the structural details from Step 3. Some examples of basic system services are

- Resource allocation
- Garbage collection
- Page swapping
- Object creation
- Exception handling

Changing product characteristics in the web-sales example requires the object-creation service to support variation. The meta-level classes for the product are instantiated with new and different attributes for each product object.

Step 5: Define the meta-level objects

In the preceding three steps, you studied the application's internals — what adapts and what remains constant. In this step, you design meta-level classes for every aspect identified in steps 2 through 4. The meta objects encapsulate the changeable attributes of the application.

The meta objects for the web-sales example are the objects that store the metadata for each product. When the product metadata is read, the meta objects that contain the product descriptions are created. When new product

attributes are added to the metadata, the meta-object creation service automatically add new attributes to the product meta objects. In this way, you can create a new type of product with an attribute combination that's never been used before.

Step 6: Define the MOP

In this step, you create the MOP, which defines how the people who adapt the application access and change it. You want the MOP to be well defined and controlled to prevent unsafe modifications of the application. The MOP must be able to change the meta objects; it also must be able to change the relationships among base-level and meta objects.

You can create MOPs in either of two ways:

- **You can integrate the protocol with the meta objects.** Each meta-level object contains the functions that are needed to change it. In many of the examples in "Looking for Reflection," earlier in this chapter, the MOP is implemented by using an editor to change the meta objects directly. In these cases, the variability of the meta objects is achieved by direct editing rather than by a system service.
- **You can implement the protocol as a separate component.** This method has the advantage of centralizing all the modifications to the application. You can reuse functions that change several meta-objects more easily, and you'll find it easier to include access controls that prevent unauthorized modification of the application.

The MOP changes only the behavior of the base-level object through the metadata that the base-level object accesses. The structure of the base-level object is not changed.

Because the product information metadata in the web-sales example is in an XML-like format, you can use an XML editor to define it. Alternatively, you can build a wizardlike application that lets the product designer create the product description and meta objects directly.

Step 7: Define the base-level objects

In this step, you design and build the core functionality at the base level. Where base-level objects need to adapt to changes in the application, the base-level objects query the meta objects for the correct parameters to use.

When you change a meta-level object, that change affects the subsequent behavior of all the base-level objects that access the meta object's variable attributes.

The base level of the web-sales example is responsible for many functions, including the following:

- Reading the metadata on behalf of the meta-object creation objects.
- Using the attributes defined in the metadata to create objects. The meta-object creation class is base-level functionality that isn't varied by the actual metadata.
- Implementing all the web-display functionality in base-level objects.

Even in an application that's very adaptable through a Reflection architecture, the bulk of the application likely won't vary and won't use reflection.

Programming Reflection Today

Reflection is all around us in modern programming languages. Using reflection used to be hard, requiring mind-twisting designs; today's programming languages, however, have built-in reflective capabilities. In the introductory paragraphs of this chapter, I say that much of the reflection that's discussed is in the form of an idiom, used to solve an individual design problem in a specific programming language. In "Designing Architectural Reflection" — and especially in "Implementing Reflection" — I describe how to use reflection as the underlying architectural backbone of an application.

In this section, I point you to a few specific examples of reflection as it's provided in programming languages today. This section is just an introduction to language-specific reflection; it starts with a popular language that has limited reflection capabilities and progresses to the more powerful functionality provided in recent languages. The capabilities discussed here are language tools that you can use to build an architecture that adapts through Reflection.

Reflection in C++

The 1998 C++ standards included several capabilities for runtime reflection, grouped under the title Run-Time Type Information (RTTI). The `typeid` operator is a useful capability that accesses the type of an object. This operator is limited to returning an object's name, which allows you to compare two objects for equality. It doesn't report whether an object is a subtype of something. Another useful capability is `dynamic_cast`. This operator has limitations much like those of `static_cast` but provides a way to do safe casting down to subclasses, which isn't possible with ordinary casting.

At compile time, the template capabilities of C++ allow the creation of classes and types that are based on the needs of the application and can adapt the application, thereby supporting reflection through recompilation.

For more information about RTTI and templates, see `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1751.html`. You can also look for libraries that extend the C++ standard to provide advanced reflection capabilities, such as the Boost Reflect library at `http://bytemaster.github.com/boost_reflect/index.html`.

Reflection in Java

The Java language reflection capabilities are supported in the `java.lang.reflect` library. Reflection is easier in Java than in C++, but it's still not full reflection because it's read-only; the Java library doesn't support modifications to the inspected code.

For more information about `java.lang.reflect`, see `http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html#package_description`.

Reflection in C#

Microsoft provides the Reflection library in C# to provide reflection capabilities. C# reflection is more powerful than Java's reflection because it allows instantiating new types at both compile time and runtime. C#'s Reflection library is described at `http://msdn.microsoft.com/en-us/library/ms173183%28v=vs.80%29.aspx`.

Reflection in Ruby

Modern, dynamically typed languages such as Ruby come with a complete suite of libraries and methods to support reflection. In Ruby, everything is an object, which causes everything in Ruby to be fair game for the available introspection and reflection methods. For more information, start at `www.ruby-doc.org/docs/ProgrammingRuby/html/ospace.html`.

Working with adaptive object models

Now that you've read about reflection in general and understand how useful architectures based on the Reflection pattern can be in a wide range of applications, I'll tell you a little about adaptive object models (AOMs). You can build whole applications with AOMs by making extensive use of the reflective characteristics of the application, which allows you to change the application's structure and behavior greatly through the use of the meta objects and MOPs.

AOMs differ from classical object-oriented (OO) systems in the following way:

- **In an OO application, you have to implement classes for all the different *things* that you need, so you have to create classes that are similar to other classes — and create many, many subclasses.** This complexity slows development and increases maintenance effort.
- **In an AOM, on the other hand, you have a (relatively) simple structure of classes that remains unchanged.** After these base-level classes are built, they stay the same. What changes in AOM is the data at the meta level. The data, when read in, causes instances of the static classes to be created to represent the data. When new data types are needed, they're just new and different instances of the type classes that were built into the static base-level application. Therefore, you can change the overall behavior of the application by creating new data types and classes with new methods and new types of data.

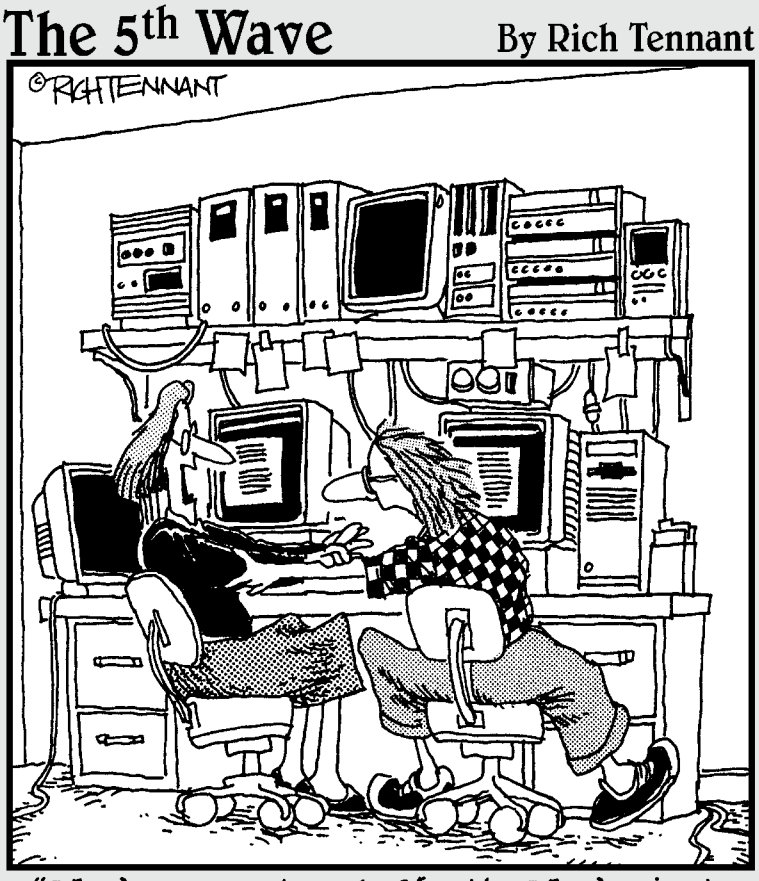
AOMs are like the Reflection pattern on steroids. They take the concept presented in this chapter to an extreme level, allowing massive reconfiguration of an application. The reflection capabilities of a programming language are used to implement AOMs.

In a typical application of reflection for idiomatic use or even as the architectural framework of a language, the percentage of code involved in reflection is low. In an AOM, however, a much greater percentage of the code is involved with reflection — usually, base-level classes checking with meta-level classes to determine parameters and configuration information.

Joe Yoder and Ralph Johnson (one of the authors of *Design Patterns: Elements of Reusable Object-Oriented Software*) present an example on their website, `www.adaptiveobjectmodel.com`. The example is an application written for the Illinois Public Health Department to track medical data. The website's articles describe how easily the application can adapt to new requirements, such as recording new types of medical test reports and accessing different databases. The application, written in the Smalltalk language, shows how thoroughly adaptive an application structured as an AOM can be.

Part IV

Designing with Other POSA Patterns



"You're a great geek, Martin. You're just not my geek."

In this part . . .

Part IV tells you about some patterns that help you solve problems you encounter in designing code. These patterns aren't designed to structure your whole system, like the patterns discussed in Part III; instead, they address design problems you may run into while implementing an architecture based on a pattern from Part III.

Chapter 17

Decomposing the System's Structure

In This Chapter

- Seeing how to create more than the sum of the parts
- Structuring your application

In this chapter, I tell you about the Whole-Part pattern, which describes how to build your system from parts so that the whole is greater than the sum of its parts. The assembled whole entity has its own properties, which are distinct from the sum of the properties of the parts. When the aggregation can do new things that aren't possible with just the parts, the resulting behavior is called *emergent behavior.* The Whole-Part pattern creates an efficient structure to unleash this emergent behavior.

The properties associated with the parts may be different from one application to another because the properties of the parts are constrained by the whole in which the parts are included.

Understanding Whole-Part Systems

A Whole-Part system has two types of participants:

- **Whole:** The *whole* is the aggregation of the smaller objects that are the parts. This grouping of parts provides a way to access part-specific functionality and provide some functionality that is available only to the whole.

A car is made from parts

A real-world example of the Whole-Part pattern is a car. The car has many parts invisible to the driver. While the car is running, fuel is being supplied to the cylinders by the fuel injector, electricity to generate a spark in the spark plug is being provided by the distributor, and they're being combined in the piston. When the spark arrives and there's fuel present, the piston will be moved to generate mechanical power. Each of these parts — fuel injector, spark plug, distributor, and piston — is a separate part within the whole of the car. The driver is the client of the system.

This whole-part design of the system simplifies changing individual parts and allows the parts to be reusable in different wholes (different cars). Each of the parts can be engineered to be very good at its responsibilities.

A whole may provide ways to access parts' functions. Also, a whole can have its own functionality, perhaps combining functionality from the parts into something new.

- **Parts:** *Parts* are contained in only — and exactly — one whole at a time. Multiple wholes can't share the same part simultaneously. Parts live only within the lifespan of a whole. In other words, parts can't exist unless the whole exists.

  Usually, parts are independent, but sometimes, they need to call one another.

For an example Whole-Part system, consider an e-mail application. This type of application is built from many parts, including the mail-receiving part; the mail-sending part; a basic display part; and parts that support many other functions, such as encryption, virus scanning, rich-text and HTML display, and message management. Some of the parts can be used individually, but some make sense only within the context of the whole application. Parts can be added, deleted, and updated with new versions.

To hold the whole together, some glue functionality is provided that provides for interaction and communication among the different parts. When combined, the parts together create a robust, secure e-mail application. An alternative solution would be a monolithic application, but maintaining such an application would be hard — and substituting new algorithms into the application would be even harder.

Seeing how the pieces fit

The parts in a Whole-Part system can fit together in one of three different ways:

- **Assembly-parts:** The relationship is between a combination of parts in some predefined way. The parts are tightly integrated. The number and type of the subassemblies are also sometimes predefined and don't vary from one assembly to another.

  Examples include a chemical molecule that has different properties from those of its components and the e-mail system described in the preceding section. A good way to implement the Forwarder-Receiver pattern that I discuss in Chapter 21 is as an assembly of parts.

- **Container-contents:** The whole is a container that holds the other parts. The contents vary in type and quantity and are loosely coupled with one another.

  An example is a package in the mail that groups its content. The package doesn't change the properties of the contents, but it may hide them.

- **Collection-members:** The aggregate is a collection of similar members, which are related in some way that the collection defines. The members are individuals but can be treated equally because there's no distinction among them.

  An example is a membership organization.

Recognizing the benefits and liabilities

Separation into whole and parts has some benefits:

- **Encapsulation of parts:** Encapsulation of parts conceals them from clients, which allows you to change the internal structure without impacting the clients.
- **Reusability:** The parts can be reused in other Whole-Part combinations.

Like all patterns, however, the Whole-Part pattern has drawbacks:

- **Slow interaction:** Interaction between the component parts and the whole entity slows things down. Unlike a monolith design, in which everything is built together, a Whole-Part design creates separate components that may be distributed.

  Communication between distributed parts will be slower than communication within a single monolith.

- **Increased complexity:** A Whole-Part system has a set of rules for how the whole and the parts relate to each other. Overall complexity increases because of these rules.

Implementing the Whole-Part Pattern

Five steps are involved in implementing the Whole-Part pattern, as you see in the following sections.

As you design the whole, you'll revisit earlier decisions and make refinements, so you should iterate and jump around during these steps rather than march through them in strict order.

Step 1: Define the whole's public interface

In this step, you define what the whole does — what the client is going to expect from the whole. In this step, you focus on the interface from the client to the whole regardless of where the functionality is provided — from the whole or from a part — and ignore functionality from the parts that are used only internally.

Step 2: Divide the whole into parts

The parts inside a whole can be derived and isolated in several ways:

- **Top-down:** In the *top-down* approach, you start with the whole and work downward, uncovering the parts that you need to implement the whole's functionality. The partitioning is driven by the services that the whole offers to the client.

  Finding all the whole's functionality can be hard if you're reusing existing components as the parts.

  The parts and the whole can become tightly coupled due to the nature of the system. The parts may not be reusable because they're designed to fit exactly what the whole needs.
- **Bottom-up:** In the *bottom-up* approach, you look for existing parts that you can reuse and couple into the whole. The parts will be loosely coupled.

  The whole may have to include glue code to bridge gaps between the existing parts.
- **Alternating:** *Alternating* between top-down and bottom-up is another effective technique. Start at one extreme and then switch to the other to ensure that the decisions you just made make sense from that direction. As the design progresses, you solve problems by alternating between the two approaches.

How you proceed in this step depends on the approach you just chose: top-down, bottom-up, or alternating. If you're alternating between bottom-up and top-down, you'll use both of those methods.

Top-down approach: Partition the whole's services

This step applies when you're working down from the top. You partition the functionality that the whole provides into the parts that can best provide that functionality. If the functionality matches an existing component or class, great! With the top-down approach, you may have to design and build new components to fill in gaps to make the whole.

You can decompose a whole into parts in several ways. A triangle, for example, can be defined by three points that aren't in a line, by three lines, or by a point and a line segment. You should select the way that makes it easiest to implement the services that the whole will provide.

Bottom-up approach: Pull parts from libraries of components

This step applies when you're working up from the bottom. Look into the class and component libraries at your disposal to identify the software parts that you'll use to make the whole. If you can't identify existing components that satisfy the entire whole's functionality, specify the additional components that you need to design, and describe how they'll interact with the other parts. You may need to use a top-down approach to implement the missing components.

Step 3: Define the services of the whole and the services offered by the parts

You can define the services offered by the whole to the client in two ways:

- The whole forwards a request for service to the part that can satisfy the request.
- The whole satisfies the request directly, but in the process, it asks the parts to supply services that the whole needs to answer the client's request.

You also can define the services in combinations of these two ways.

Further, you have two ways for the whole to send requests to the parts:

- The whole can make a request to a part, using the part's interface. In this case, the part doesn't know anything about the context of the whole; it responds to the request using only its own environment. This design leads to loose coupling of the components because they don't rely on one another's environments.
- The whole can ask the part to respond to the entire request, which increases coupling between the part and the whole. The whole must send enough information about its context to the part for the part to be able to respond to the request on behalf of the whole.

Using the Composite pattern

The Composite pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional), is a variation of the Whole-Part pattern that is applicable when you have these two requirements:

- You want to represent the Whole-Part object hierarchies.
- Clients shouldn't care about whether something is a composite of objects or an individual object; they should treat composites and individual objects the same way.

Step 4: Build the parts

You may need to design the parts recursively if they're also Whole-Part combinations. This arrangement is actually quite common. Consider an example: A bicycle (whole) is made of wheels (parts), which are themselves wholes made up of other parts (hubs, spokes, rims, tires, and so on). Start at Step 1 to divide these new wholes.

Step 5: Implement the whole

The whole manages the life cycle of the parts, so in this step, you need to implement the mechanisms to create and delete parts.

Implement the whole's services, building the services that depend on services from the parts. You also need to implement services that don't invoke any of the parts and that are self-contained within the whole.

There may be constraints on the whole, such as behavior of the whole that differs from the possible behaviors of the parts. Also, the constraints may relate to the parts themselves. The sizes of the parts, for example, can't exceed the size of the shipping-container whole.

You may find that you alternate between designing the wholes and designing the parts as you create the total solution.

Chapter 18

Making a Component the Master

In This Chapter

- Making runtime more efficient with divisible tasks
- Putting the Master-Slave pattern to work

In this chapter, I tell you about the Master-Slave pattern, which is useful for dividing work among processing elements to improve performance or reliability. The responsibilities of the master and the slave are well-defined and not interchangeable; all the slaves are doing identical or comparable work. The slave's roles are firmly defined before execution and need to be coordinated — a requirement that differentiates this pattern from the general problem of dividing work and scheduling it, which the operating system normally solves.

Introducing the Master-Slave Pattern

All the subtasks in a Master-Slave pattern are identical, so this pattern helps you coordinate the work at runtime efficiently. This pattern differs from the Whole-Part pattern (see Chapter 17), which helps you create an efficient structure of the parts of your system during development. Figure 18-1 shows the structure of the Master-Slave pattern.

A good division between master and slaves is transparent to the clients that request the master to perform the task. The clients shouldn't be aware that a divide-and-conquer approach is being used.

The slaves that are doing the work shouldn't have any dependencies that affect that the way the work is divided. Dependencies make the solution inflexible and make it harder to achieve good separation between master and slaves.

Taking the census

Every ten years, there's a census in the United States. The primary client that uses the data is Congress. They commission the Census Bureau to take the census. The Census Bureau hires many individuals to go out across the country to count the people. The census takers all do the same things — they're interchangeable, and if one finishes in his territory, he can be moved to help out in another territory.

The Census Bureau is the master of the Master-Slave architecture. The slave role of Master-Slave is played by the census takers. The role of client is played by Congress, which authorizes the infrastructure to conduct the census.

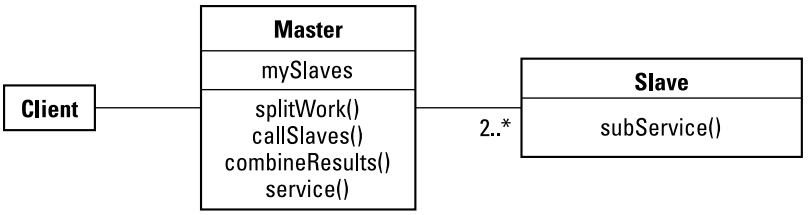


Figure 18-1: The Master-Slave structure.

Sometimes, the master must provide a coordination function for the slaves, so in addition to distributing the work, the master may have to pass data between the slaves or collect and process the results from the slaves.

The Master-Slave pattern is useful in a couple application areas:

- **Fault tolerance is enhanced through multiple computations that are compared and factored into the "correct" response.** You can use the results of the slaves to check one another for correctness by using different but semantically equivalent approaches as the individual parts. The master contains the voting algorithm that triggers the computation, compares the results, and selects one.

  NASA used this approach, which is very useful in fault-tolerant systems, to coordinate the space shuttle's main computers.

- **In parallel computing, the Master-Slave pattern spreads the workload across multiple processors.** This is how Google's Map-Reduce works, by spreading the work across multiple slaves.

Benefits of Master-Slave

The Master-Slave pattern provides some specific benefits:

- **Exchangeability and extensibility:** If you have an abstract slave class, you can change the slave implementations to substitute different implementations or to extend the capabilities without making major changes in the master. This benefit is related to the Strategy pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).
- **Separation of concerns:** Master-Slave helps separate concerns by separating the work done by the slaves from the management done by the master.
- **Distribution of workload:** Master-Slave spreads the workload and allows highly parallel processors to function efficiently, increasing the power of parallel processing.

Liabilities of Master-Slave

Like all patterns, Master-Slave has some liabilities:

- **Difficulty of division:** Dividing an application into the slave parts that can be executed in parallel isn't always feasible.
- **Hardware dependency:** Implementations of Master-Slave for parallel-processing efficiency can become dependent on the structure of the hardware that is used.
- **Difficulty of implementation:** Master-Slave is hard to implement well. Implementing this pattern involves many considerations, such as how the task is divided up, how the master and slaves collaborate, and how the final result is computed. (See the following section for more information on this liability.)

Implementing Master-Slave

Implementing Master-Slave is straightforward. Before starting, however, decide how you'll divide the work among the master and slaves. You also need to answer a few questions that will guide your implementation:

- **Will the master and slaves be in separate processes or separate threads, or will you let the operating system decide for you?** The latter method is called *black-box execution;* you don't know what's being used. The answer to this question influences the mechanisms that the master uses to communicate with the slaves.
- **If your application requires coordination of the slaves, will that coordination be done by the slaves themselves or by the master?**

Step 1: Divide the work

Define how the computation can be split into equal subtasks. You could base the split on the memory size of the task, for example, or the expected execution time. Sometimes, the work is divided based on the number of elements to be analyzed by the subtasks.

When defining subtasks, consider the environment that will process them. It's possible to define tasks that are too fine-grained for some processor architectures and that require extra overhead at the master level to manage the subtasks.

Step 2: Combine the subtasks

In Step 1, you decide how the work is going to be defined. In this step, you decide how the results from the subtasks will be combined.

Step 3: Define how master and slaves will cooperate

This step defines an interface for the task division you identified in Step 1. Subtasks can be passed to slaves as calls or parameters, or they can be placed in a repository that contains task assignments for the slaves to access. Similarly, the responses from the slaves can be in the form of parameters or function calls, or the results can be placed in a repository.

In this step, you decide how to handle the data needed by the slaves. Slaves can use shared data structures, or each slave can have its own data structures.

Factors to consider in deciding on an approach are the costs of passing subtasks to slaves, duplicating data structures, and creating shared data structures. Another thing to consider is whether slaves modify the original data. If they do modify the data that other slaves are sharing, they need their own copy of the data.

Step 4: Implement the slave components

Build the actual slaves to perform the subtasks from Step 1 with the interfaces you defined in Step 3.

Step 5: Build the master component

In general, tasks can be divided into a fixed number of subtasks. Master-Slave is most applicable when a complete task is handed off to the slaves for processing. This helps increase the fault tolerance of an application. An application gets difficult to build, test, and maintain when the master performs part of a task and the slaves perform the remainder.

Another option is to divide the overall application into as many tasks as possible. This is especially useful when dividing work over a large number of processors.

The master must have the code it needs to start the slaves, manage their processing, collect the results, kill the slaves, and combine the results from the many slaves into the final product. The master also must handle errors — such as failures of slaves or failures of threads — and provide for graceful handling of the errors.

Some new programming languages, such as Erlang, are ideally suited for Master-Slave parallel processing.

Chapter 19

Controlling Access

In This Chapter

- Seeing why proxies are useful
- Recognizing different kinds of proxies
- Adding a proxy to your system

In this chapter, I tell you how to isolate a component with a proxy — specifically, the Proxy pattern. You use this pattern when a client needs the services of another component and direct communication — even though possible — isn't the best solution. The proxy component provides an indirect way to access a component to coordinate, filter, and control direct access. This may be done to increase security, enforce protocols, or speed up operations, among other things that I cover in the "Getting Acquainted with Proxy Variants" section.

Many people, including Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional) have written versions of the Proxy pattern. In this chapter, I discuss a general version of Proxy. You may encounter many variants that refine and adapt this general version.

Understanding Proxies

Proxies are needed when it's inappropriate for two components to communicate directly. This can happen when the components shouldn't know where the others exist in a distributed system or when address information shouldn't be hard-coded into the components — even if the components are local to the same processor.

The proxy should be efficient in terms of execution time; it should be able to determine quickly which component should be accessed and identify its location. Notwithstanding the reasons *not* to connect the client and server directly, the connection through the proxy should be transparent to the client; the client shouldn't realize that it isn't talking directly with the server.

Proxies benefit their applications by decoupling clients from the locations of the servers that they use. Proxies also help you structure the components to separate the required client-server interaction from the housekeeping associated with the client's finding and referencing the location of a server.

Because the proxy solution is transparent to the client, however, it can be inefficient. The client must know the differences in costs between local and remote services, and use the proxied remote service only when appropriate.

Don't make the strategies for caching or loading on demand discussed in some of the variants too complex. Doing so will increase development and maintenance costs, as well as introduce more places for bugs to hide.

The Proxy pattern versus the Broker pattern

At first glance, the Proxy pattern is similar to the Broker pattern (see Chapter 12) because both serve as intermediaries between clients and servers. The difference is that in a Broker architecture, a client requests that the Broker component find some server that can provide a specific service to the client; the choice of server is left to the broker. A proxy serves as a representative for the server after the server's address has already been found. The client either makes the server selection itself or asks a broker to make a selection.

Parts of a proxy

A proxy has four parts:

- **Server:** The server component is the original server component that will respond to the client's request. Servers can perform everything from simple functions such as simple data responses to complex functionality that involves other components.

- **Proxy:** The proxy component stands in for the server and is the server interface for the client because it has the same interface as the server. There's usually a one-to-one relationship between the proxy and the server — although not always, as you see in "Getting Acquainted with Proxy Variants," later in this chapter.
- **AbstractServer:** If the proxy is to provide exactly the same interface as the server, you should define the interface in the AbstractServer class and then inherit the interface in both the server and proxy classes.
- **Client:** The client is the component that wants the server to provide some service. It can communicate with the server directly, but for the reasons explained in this pattern, it interfaces with the proxy instead.

Figure 19-1 shows the relationships among these parts.

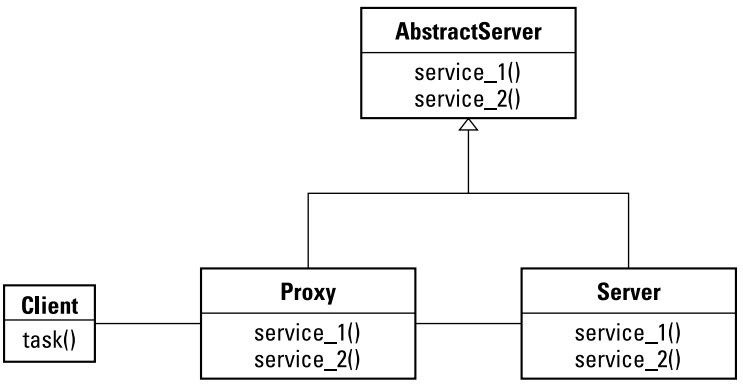


Figure 19-1: Proxy class diagram.

E-mail forwarding

An e-mail proxy is one of the common real-world examples of the Proxy pattern that you run into without realizing it. An e-mail proxy will receive an e-mail for a recipient, look up his real e-mail address, and forward the mail to the real e-mail address. These proxies are available from services like Pobox (`www.pobox.com`) and many professional organizations like the IEEE (`www.ieee.org`) and the Association for Computing Machinery (ACM; `www.acm.org`).

Getting Acquainted with Proxy Variants

Proxies are all around us; they're involved in most communication exchanges with web pages and throughout other non-Internet applications. Proxies come in many flavors, as I discuss in this section.

Remote

A *remote proxy* hides the physical location of the server. It implements the needed interprocess communication (IPC) to allow the client to interact with the server. You can further encapsulate the IPC by providing Forwarder-Receiver elements to the client and server, as I explain in Chapter 21.

Protection

You use a *protection proxy* to protect the server from the client. The proxy checks the access rights of every client that attempts to talk to the server. A common implementation of this kind of proxy is an access-control list.

Cache

In a *cache proxy,* the proxy is given a data area where it can hold results to speed repetitive accesses to the server. The cache proxy requires a strategy for various aspects of its work, such as what results from the server to cache, how often to refresh the cache, and what to do when the cache is full. The strategy must also account for cache invalidation when the server results have changed from what is stored in the cache. Web browsers are almost always cache proxies.

Synchronization

When it's important for only one client (or some finite number of clients) to access the server simultaneously, you should use a synchronization proxy. A *synchronization proxy* provides mutually exclusive access to the server. Sometimes, you want the proxy to distinguish between read and write accesses to the server and to coordinate those accesses differently.

Counting

You use a *counting proxy* to collect use statistics or to tell the system how many clients are interacting with the servers. You can also use this type of proxy to determine when the server is no longer needed (the number of clients becomes zero) and can be deleted.

The Counted Pointer idiom, discussed in Chapter 22, describes a way to implement a counting proxy in C++.

Virtual

A *virtual proxy* (also known as a *lazy constructor*) is used to hide the fact that the server, or the server's storage, isn't fully instantiated. The instantiation and loading of the missing parts of the server are done on demand.

When a request for service arrives, the virtual proxy decides how to proceed. If the server is fully available, the proxy merely forwards the request to the server. If the server isn't fully instantiated, the proxy triggers the server's start-up process or directs the server to become more fully instantiated, maybe by loading more of its data into memory.

Firewall

A *firewall proxy* provides secure client and server communication when the environment contains threats. Firewall proxies are usually implemented as daemon processes on separate computers, which can be called proxy servers. All client requests directed to the outside world pass through the firewall proxy. The proxy checks requests and incoming replies for compliance with security and access policies. If a request or reply doesn't conform to the policy, that request or reply is blocked.

In a well-designed firewall proxy, the clients and servers shouldn't be aware that the proxy is filtering their messages.

Because all communications flow through the firewall proxy, it has the potential to become a bottleneck.

Reverse

A *reverse proxy* operates on behalf of a server and provides capabilities such as load balancing, caching, and authentication. The reverse proxy appears to the clients as a single server but, in fact, is serving as a proxy for multiple servers. Reverse proxies can provide the same kinds of functionality as the other proxy variants, only they primarily control a server's access to the rest of the system or Internet.

Implementing a Proxy

Implementing a proxy is straightforward. Depending on which proxy variant you're implementing (see the preceding section), you'll customize the steps to the variant.

Step 1: Identify access control responsibilities

Start by identifying all the access responsibilities of the server and assigning them to the proxy component.

Step 2: Introduce an abstract base class

When your language supports inheritance, it's useful to introduce a base class, AbstractServer, described earlier in this chapter and shown in Figure 19-1. Both the server and the proxy inherit the access responsibilities from the abstract base class.

You can use the Adapter pattern (from *Design Patterns: Elements of Reusable Object-Oriented Software*) to adapt among the interfaces if the server and the proxy can't have identical interfaces that they inherit from the abstract base class.

If your language doesn't support inheritance, consider defining the interface in a library or module that can be shared to help keep the interfaces all the same.

Step 3: Implement the proxy's functions

In this step, you implement the responsibilities that you identified for the proxy in Step 1. You implement both the access functionality that you're removing from the server to place into the proxy and also the mechanisms for the proxy and server to communicate. You can implement the proxy to server communication using the existing server interfaces, or define a new mechanism that may improve performance or include server control functions.

Step 4: Remove responsibilities from the server

The proxy assumes the access responsibilities of the client and the server. In this step, you reallocate these responsibilities from the client or server and give them to the proxy.

Step 5: Give the proxy the address of the server

This step associates the proxy and server so that the proxy can pass requests to the server. The handle can be whatever is convenient: a pointer, a memory address, a socket, a service ID, or whatever makes the most sense for the system.

Step 6: Remove the relationships between the clients and servers

All communications among the clients and the servers should go through the proxy. Remove any direct relationships and linkages, and replace them with linkages to the proxy. The proxy is connected to the server in Step 5.

Chapter 20

Managing the System

In This Chapter

- Processing commands
- Managing multiple views

When I was working on my first object-oriented project as part of a team, our consultant told us that if we identified an object with the name "something *manager,*" we should beware. The problem is that manager objects can start to take on more and more responsibility, making the objects that are managed very lightweight — and preventing a clean object-oriented design.

But creating objects that will "manage" other objects helps when the other objects to be managed are similar and perform the application's real work. In the two design patterns in this chapter, the collections of objects are *commands* and *views.*

Keep your manager objects simple and their responsibilities truly managerial. Let the objects that they manage do the actual work.

This chapter describes two patterns that you can use to manage different situations. Command Processor, the first pattern, helps when you're implementing an application that has user commands. The second pattern, View Handler, aids your application when it has different displays to handle — like the Model-View-Controller (MVC) and Presentation-Abstraction-Controller (PAC) architectures (see Chapters 13 and 14, respectively).

Separating Requests from Execution with Command Processor

When you're building a system that has multiple commands, the system can become unwieldy if each command is handled separately. When a command is triggered by an external (or internal) event, some large switchlike functionality usually routes the event to the correct command. When new commands are added to the system, both the new command code and the event-routing functionality must change. It's even worse when the arguments or parameters of a command change, because then you must find every place in the code that refers to the command and change it — a maintenance headache.

The Command Processor design pattern offers a better, easier way to add or delete commands. A ComProcessor is the manager of the actual commands and oversees their functions; it also provides the framework for including undo mechanisms.

Looking inside the pattern structure

Five classes are involved in the Command Processor design pattern:

- The AbstractCommand superclass and the ComObj classes implement the interfaces of the command.
- The Controller and ComProcessor classes manage command execution.
- Individual objects of the fifth class type, Supplier, do the actual work required for the commands to succeed.

Figure 20-1 shows how all these classes fit together into the system.

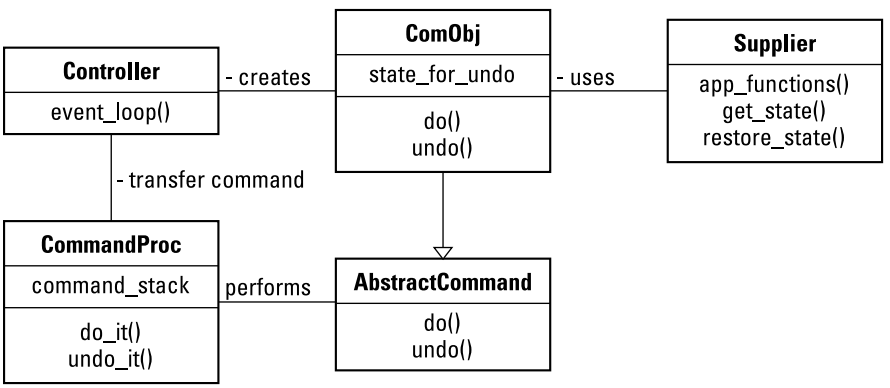


Figure 20-1: Class diagram for the Command Processor pattern.

Making a family of command classes

The AbstractCommand class is in the system to give every command a uniform interface. At its simplest, it defines a method for executing a command and another for undoing that command. Depending on your application, it also may define the interfaces for other functions, such as logging.

The ComObj class inherits the basic invocation methods from the AbstractCommand class. A ComObj exists for each command the user may want to invoke. When the ComObj is started and initialized, the parameters that define the user's specific needs — such as the starting and ending locations of text to be deleted — are given to the ComObj through its initialization code.

The Supplier classes are helpers for the ComObjs. They perform the application-specific functionality needed for a ComObj to perform its function. Suppliers can be shared by several ComObjs.

Structuring the manager classes

The Controller class is the interface between the command-management part of the system and the event-processing part of the system. The controller is responsible for accepting user requests from the event-processing system and instantiating new ComObj objects to perform the request. Any parameters provided by the event are used when the ComObj is created. The controller doesn't start the ComObj executing the request; instead, it passes the new ComObj object to the ComProcessor.

An instance of the ComProcessor class receives a ComObj object from the controller and starts ComObj execution by invoking the `do` method that the ComObj inherited from the AbstractCommand. The ComProcessor doesn't become involved in the particulars of the ComObj or its execution; it only uses methods that are defined for all the commands by the AbstractCommand class. The ComProcessor can optionally be given other intelligence about the problem so that it effectively schedules ComObjs based on threads; processes; time of day; or criteria other than first-come, first-served.

Watching it work

Figure 20-2 shows how everything works together. The process involves six steps, indicated by circled numbers in the figure:

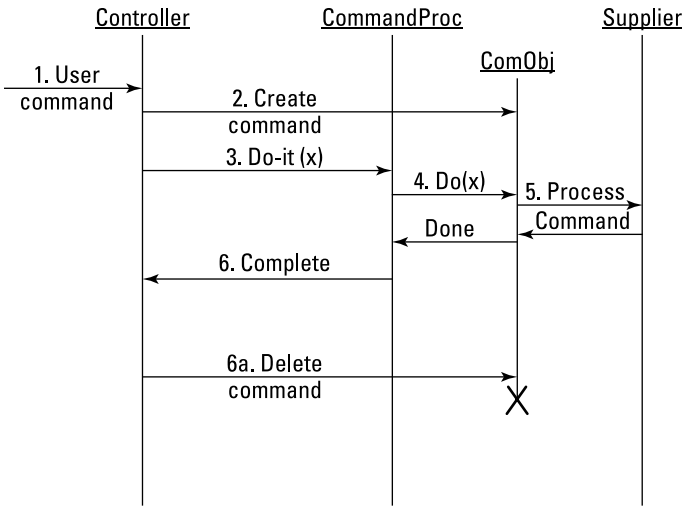


Figure 20-2: A Command Processor pattern scenario.

1. **The user invokes a command.**

   The event specifying this command is sent to the controller.

2. **The controller determines which subclass of AbstractCommand satisfies the requested user command and creates the ComObj object, giving it the parameters that came as part of the event.**

3. **After the ComObj is created, the ComObj is passed to the ComProcessor through the ComProcessor's `do_it` method.**

4. **The ComProcessor invokes the ComObj's `do` method.**

5. **The ComObj takes whatever action is necessary to satisfy the user's request.**

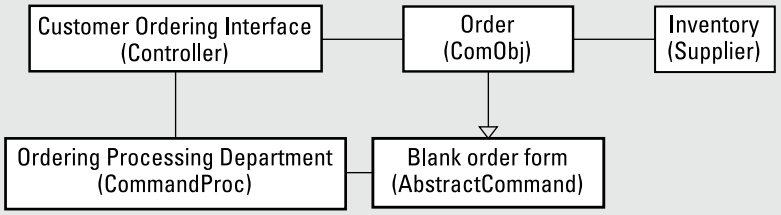
   The ComObj may involve Supplier classes to satisfy the request.

6. **The ComProcessor reports back to the controller that the ComObj is complete.**

   In some situations (shown as message 6a in Figure 20-2), the ComProcessor also triggers the deletion of the ComObj object; at other times, it keeps the ComObj alive so that it can be undone. The choice depends on the application.

Orders as commands

When you order something from a catalog, you talk to a person or system that receives and processes your order. The customer interface (person or web page) corresponds to the controller. Orders are fulfilled by a department that corresponds to the CommandProc. The inventory of items to be sold that are sitting on a shelf correspond to the supplier. The actual order corresponds to the command (the ComObj). The actual order started as a duplicate of the AbstractOrder.



Creating undo mechanisms

Frequently, a user will want to be able to undo commands. This pattern provides an undo mechanism through the ComProcessor, AbstractCommand, and ComObj classes. Each ComObj that can undo its work is created with an `undo` method that performs the task. The AbstractCommand provides the method definition. The ComProcessor saves the ComObjs in a stack after they're executed. When an undo request is received, the ComProcessor invokes the `undo` method of the ComObj to be undone.

Creating an undo mechanism can be hard. Sometimes, functions can't be undone (such as when converting files from one type to another); at other times, the state needs to be saved to have an undo make sense (such as during a multiple-screen web transaction).

Implementing Command Processor

Creating a basic Command Processor structure involves four steps, which I describe in the following sections.

Step 1: Define the AbstractCommand interface

The first step is defining the interface of the AbstractCommand class. At a minimum, this class contains a `do` method that executes a ComObj. The interface also may include an `undo` method and other methods that invoke functionality specific to the application, such as specialized methods to retrieve relevant filenames.

Step 2: Design the ComObj components

After you create the interface of the AbstractCommand class, your next task is creating the individual ComObj components. The ComObjs substitute concrete methods for the abstract ones defined in the AbstractCommand. The ComObj classes perform — or ask Supplier classes to perform — the desired user functionality. The choice of Supplier classes to engage can be done statically during development or dynamically, based on the initialization of the ComObj object.

The number of commands (and ComObjs) can explode in a command-rich application. To reduce the number of commands, have ComObj objects interface to groups of actual commands. You can organize the groups by abstraction or by the Suppliers used by a task, or you can preprogram combinations of commands together into a higher-level command.

Step 3: Build the Controller component

The Controller creates the ComObj objects based on the events received. It can be helped by creational patterns such as Abstract Factory and Prototype from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).

Step 4: Build the ComProcessor

The ComProcessor receives a ComObj from the Controller and takes responsibility for it, invoking the ComObj's `do` method. The ComProcessor also may be responsible for deleting the ComObj after it has completed or for saving it to invoke its `undo` method later.

The Command pattern from *Design Patterns: Elements of Reusable Object-Oriented Software* is similar, but here, I give you more details about how you can build a system with the concept.

Managing Your Views with View Handler

Sometimes, the objects that your application needs to manage are multiple views. The views might be views in the MVC architecture (see Chapter 13) or intermediate-level agents in a PAC architecture (see Chapter 14). This pattern can help with views in either of those architectures, but it isn't limited to those two patterns. Here's how the View Handler pattern helps them:

- **Model-View-Controller (MVC):** The View Handler pattern refines and explains the relationship between the model and related views.
- **Presentation-Abstraction-Control (PAC):** View Handler is important in PAC for coordinating multiple views. Intermediate-level PAC agents correspond to View Handler because they manage view elements. The bottom-level PAC components — components that are primarily involved with views — represent the view components of the View Handler pattern.

There are several reasons to add a manager for the views:

- The interactions with the views should be handled consistently so that the system is easy for both users and software clients to use.
- You don't want to merge the code for each view, because it's independent of the other views' code.
- Tying the code for the views to the management code isn't good. In systems with multiple views, the system should make adding new views in the future easy.

Looking inside View Handler

The View Handler structure involves four types of classes:

- *ViewHandler* is the most important class discussed here.
- Views are implemented through the *AbstractView* superclass and the *SpecificView* classes.
- Finally, *Supplier* components provide the views of the data that they display.

The ViewHandler class is responsible for opening, manipulating, and closing the views of the system. When a user wants to create a view, the ViewHandler creates the SpecificView responsible for the view; similarly, when the user deletes a view, the ViewHandler does the deletion. The ViewHandler also

directs the SpecificViews when the user makes requests such as changing the visibility of the views or resizing them. The SpecificViews are coordinated by the ViewHandler when they must exchange information or be updated after some data changes.

The ViewHandler interacts with SpecificViews through the interface defined by the AbstractView class, which defines the interface for everything a SpecificView can possibly do.

The SpecificView classes inherit the interface from their AbstractView superclass. Each SpecificView has its own display function that gets data from its Suppliers and presents the display to the user. Instructions for manipulating the SpecificView's display come from the ViewHandler through the AbstractView interface.

You can see the structure of the View Handler pattern in Figure 20-3.

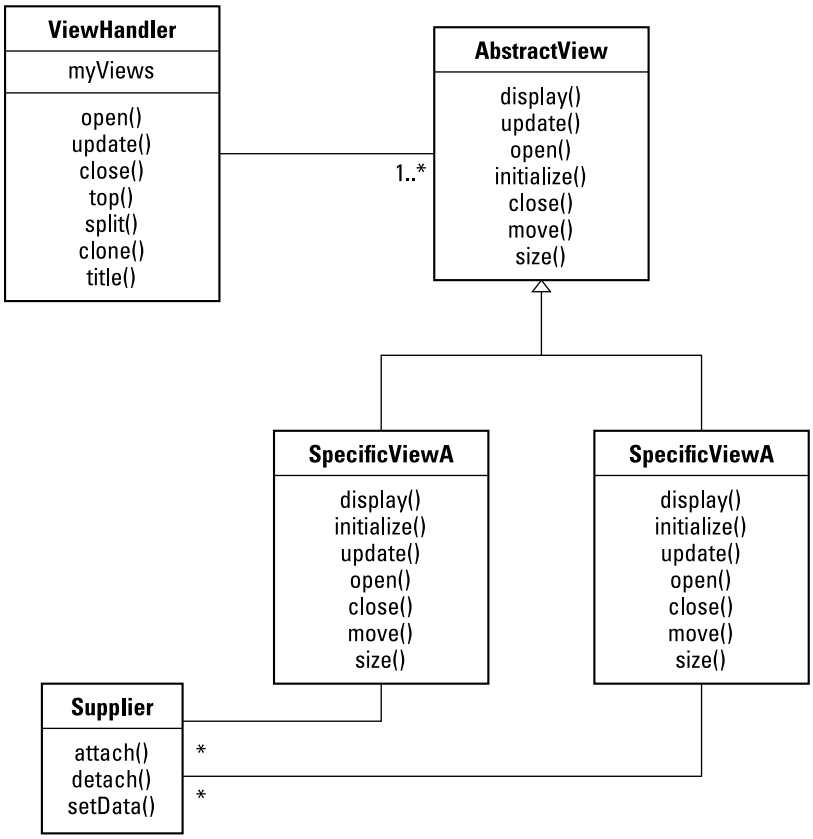


Figure 20-3: View Handler class diagram.

Managing your desktop windows

The real-life example of the View Handler pattern is the windowing system on your computer. It manages the windows, which are the views of your programs that you want to see. The underlying programs and files correspond to the suppliers. The windowing system provides a consistent and uniform handling of the views and seamless creation of new views.

Implementing View Handler

Assuming that the Suppliers of the data to be displayed in the views already exist, four steps are involved in implementing a View Handler structure, and I discuss them all in the following sections.

Step 1: Identify the views

In this step, you define what views the system will display, as well as the ways in which a user can interact with and control each display.

Step 2: Define the view's common interface

In this step, you examine the views that you identified in Step 1 and define a common interface for all the views by defining the interface of the AbstractView class. At minimum, the interface must include methods to create, open, and close a display, as well as methods to update a display.

Step 3: Implement the views

Each view has a SpecificView, which is a derived class from AbstractView and which implements the interface defined in AbstractView. In this step, you design the parts of a SpecificView that make it unique. The SpecificViews take data provided by their Suppliers and create the display.

When a SpecificView makes a change that affects the other views, the SpecificView must notify the ViewHandler, which notifies the other SpecificViews. This situation may arise when a view is moved or resized, thereby changing the visibility of other views. The Publisher-Subscriber pattern (see Chapter 21) can help with this notification.

Step 4: Define ViewHandler

In the last step, you implement the code that creates and initializes SpecificViews. You can use the Factory Methods pattern from *Design Patterns: Elements of Reusable Object-Oriented Software* for this purpose.

The ViewHandler keeps track of all the open SpecificViews and may also keep track of other information, such as current size and position. Sometimes, the ViewHandler contains application-specific coordination functionality. Examples of this functionality include paired views in which displaying one view should also display another view or when one view presents information about another view.

Chapter 21

Enhancing Interprocess Communication

In This Chapter

- Acknowledging problems with distribution
- Separating the components for communication
- Using a dispatcher
- Publishing change notifications and updates

You need to solve many problems when you build distributed systems — portability, modularity, location transparency, and consistency of communication among components. This chapter presents three design patterns that help you overcome these challenges.

In the first two sections, I tell you about two patterns that address the problem of location transparency. Forwarder-Receiver tackles the issues of transparency and portability by encapsulating the details of communication into specific components. Client-Dispatcher-Server adds an intermediate layer to connect the client and server components transparently.

In the last section, I discuss the Publisher-Subscriber pattern, which helps you address the component-consistency problem. It's closely related to (some people think indistinguishable from) the Observer pattern in *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).

I don't focus on the details of the communications, such as what protocols are actually in use, because these patterns apply to any kind of interprocess communication (IPC) method that you choose.

Forwarding Messages to a Receiver

It's very common nowadays to build systems that have client components communicate with server components to get something done. Even if the systems don't have explicit clients and servers, multiple peer components work together to provide the same functionality.

The quick, easy way to implement client and server communication is to have the client component send requests directly to the server. This method is also the most efficient in terms of execution overhead because no other steps or parties to the communication are in the way to slow it down.

If you connect clients and servers directly, one temptation is to intermix the communication code with the client and server functionality. This mixing, however, becomes a headache when the time comes to update either the client or the server code, because you need to find each and every place where you put direct communications.

Using specialized components

The Forwarder-Receiver pattern offers a way to avert maintenance headaches and improve overall encapsulation in your code with only a very minor effect on efficiency. Figure 21-1 shows the basic structure of this pattern. As you see in the figure, each client and server component (these components are called *peers* because they work together) has two subcomponents: a *forwarder* and a *receiver.*

Forwarding messages

When you were a schoolchild, you probably passed messages from one of your classmates to another. You were serving as a peer, handing messages between your classmates. The one who sent the note acted as a *forwarder* sending the message to the *receiver.* You served as the communications medium. The forwarder didn't control how the message was routed toward the intended recipient. This mechanism is efficient and easily handles changes in the message format — for example, if the message were written in Chinese or Italian.

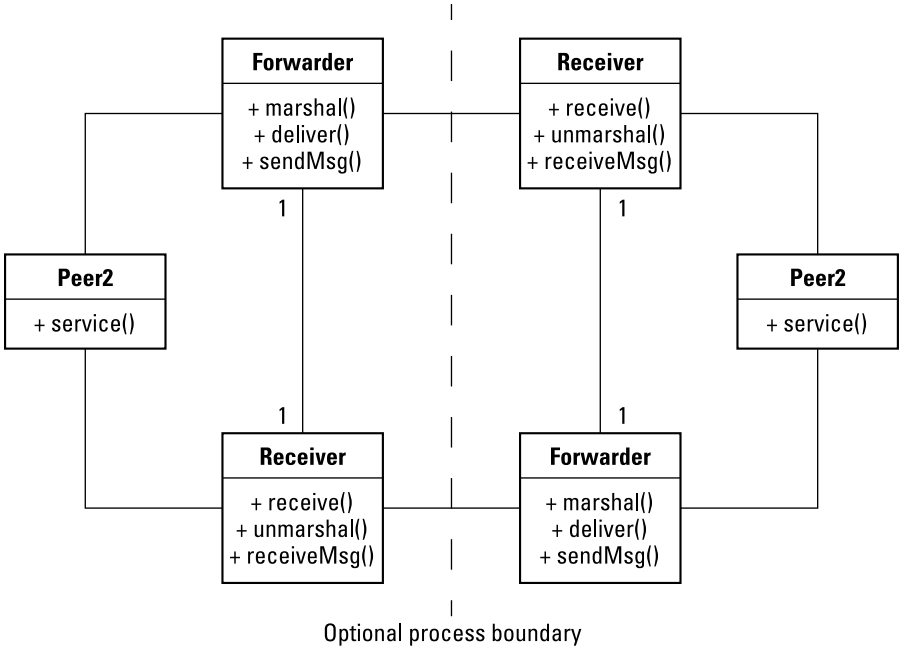


Figure 21-1: Forwarder-Receiver design structure.

The peers provide the general application functionality required to communicate with other peers, as follows:

- **Forwarder:** The forwarder sends messages from the peer to another peer. It packages the message, which might require serializing an internal data representation. The forwarder locates the peer that should receive the message by mapping a name to a physical address and then sends the message to that address. The forwarder has other responsibilities, too — encapsulating the details of the IPC mechanism used.
- **Receiver:** The receiver watches a physical address for a message to arrive. When a message arrives, the receiver unpackages the message — deserializes it, if necessary — and delivers it to the receiving peer.

Figure 21-2 shows a message-sequence diagram for this pattern. As you can see in the figure, the forwarders only send messages outward from a peer, and the receiver only receives messages sent to a peer. These two separate components have specialized responsibilities.

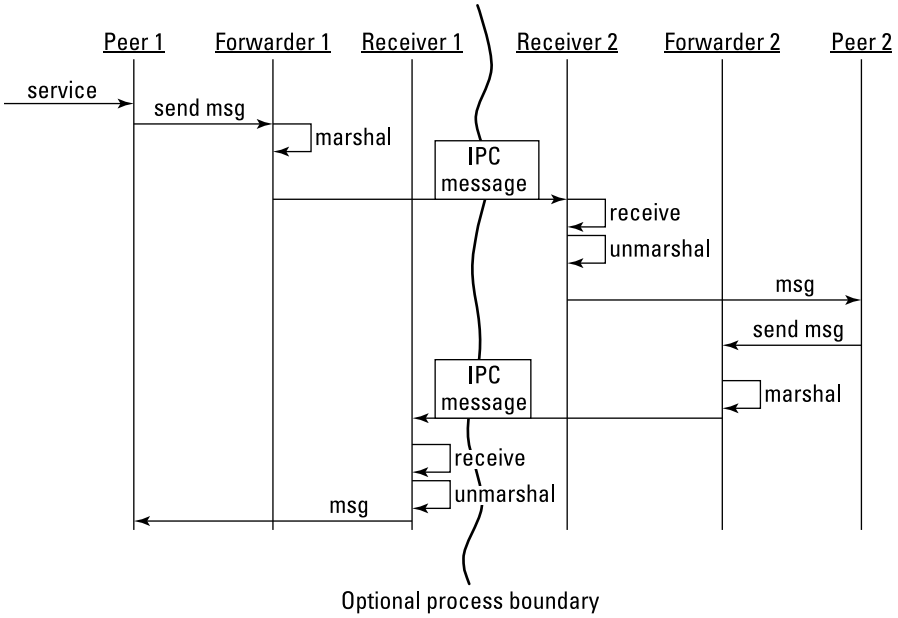


Figure 21-2: Forwarder-Receiver message-sequence diagram.

When maintenance is required — when peers change or a different kind of message is required, for example — it's simple because all the communication functions are contained in the forwarder and receiver components. You won't have to search high and low through the code for places to change.

Implementing Forwarder-Receiver

Implementing this pattern involves six steps, which you should iterate through.

Step 1: Define the name-to-address mapping

This pattern allows peers to reference other peers by name. The first thing you must do is decide the names of the peers. To do so, you define a *namespace,* which prescribes the rules for naming and any constraints on the names. Here are some example naming rules, with examples:

- Names are URLs (`ipc://server/service`).
- Names are IP addresses and port numbers (`192.168.1.112:4110`).
- Names start with a capital letter and are exactly nine characters long (`MyPeerSvc`).

- Names start with a capital letter to encode a five-character location abbreviation, followed by a number identifying the specific peer (`ChiI143`).
- Names have Unix- or Windows-like addresses (`/Server/Videoserver/AVIServer` or `\\Server\AVIServer`).

Forwarders send messages to a particular address. More than one receiver can be listening to an address, which allows you to broadcast the message to several recipients.

Step 2: Define the message protocols

Forwarders and receivers must communicate with the peers that they represent. You need to define the detailed protocol to be used. This protocol is internal to a side of the communication, such as Peer 1 or Peer 2 in Figure 21-1 or Figure 21-2. You should use the same protocol from peer to forwarder and from receiver to peer.

The messages between the forwarder and receiver in different peers need to be in a format that the forwarder and receiver both recognize. In this step, you define the detailed structure of the messages that will flow between forwarder and receiver.

Your protocols must define appropriate behavior when communications *time out* — that is, when replies aren't received within the required period. You also must consider the behavior of the protocol when communication fails. Possible actions to take include retransmitting messages and reporting exceptions.

Step 3: Choose a communication mechanism

This step focuses on the forwarder-to-receiver communication path and specifies how communications will occur. Your choices will be guided by what's available in your operating system.

If efficiency is important, you can use a low-level mechanism like TCP/IP. It's efficient and flexible, but programming it can be difficult. If your application needs to be portable because you'll move it to different operating systems, consider a mechanism like sockets. Sockets are available for most operating systems now, are sufficiently efficient for most applications, and aren't quite as closely tied to operating-system functions as TCP/IP would be.

Step 4: Build the forwarder

In this step, you build the forwarder that receives messages from the peer, packages the message, determines the recipient, and sends the message via the mechanism you chose in Step 3.

The forwarder uses an internal mapping of name to physical address. The *name* is what the peer knows the other peer by, and the *physical address* is the destination in the communications path to the receiver. The mapping can be defined statically before execution, or it can be adjusted dynamically at runtime.

Another option you must consider is whether each forwarder will have its own mapping repository or whether several forwarders will share a single repository. Maintaining the repository is simpler if it's shared across the application, but flexibility is enhanced if each forwarder each has its own repository, because different forwarders can map the same peer address to different distant components, which can help with distribution for performance or fault tolerance.

You can use the Whole-Part pattern from Chapter 17 to divide the structure of your program to provide the repository, receive the message from the peer, and send the message over the communications channel to the receiver. This pattern increases forwarder encapsulation and maintainability.

Step 5: Build the receiver

The receiver component receives a message from the communications channel, deserializes it for the peer, and then passes the message to the peer. The Whole-Part pattern (discussed in Chapter 17) makes the receiver's structure more maintainable, just as it does for the forwarder's structure.

During this step, you must make an important decision about the blocking behavior of your receiver. Because the peers aren't synchronized, you must decide whether the receivers should block while waiting for a message to arrive, as follows:

- **Blocking:** If the receiver waits for a message to arrive, the peer is blocked from doing any other work until the receiver hands control back to the peer. This behavior is appropriate if the peer depends on the arriving messages to proceed.
- **Nonblocking:** If the peer has other work that it can do between processing of messages, it should be nonblocking. The receiver is given a time-out value when the peer asks for messages. The receiver waits for a message to arrive or the time-out to be exceeded; after the wait, it returns processing to the peer.

If nonblocking communications isn't supported, you can place the receiver in a separate thread to provide that behavior. Only the receiver thread will block.

Another aspect of the receiver to consider is whether more than one communications channel is supported. You can multiplex communications by allowing the receiver to monitor several channels. When a message is received on any of the channels, it's passed to the peer. When more than one message is received simultaneously, the receiver must implement an internal buffer or message queue to collect one message while another is being deserialized and passed to the peer. You can use multiple threads to simulate the behavior of multiplexing by giving each thread responsibility for a particular channel.

Check out the Reactor pattern in *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley), for more information about demultiplexing events.

Step 6: Create the application peers

You may have your application software built before adding forwarders and receivers. In this step, you must separate the peer software into two parts: the client and the server. Some parts of your application can be both clients and servers — in other words, both requesting services from other servers and providing services to other peers that are playing the client role.

A client sends a message to the remote peer and waits for a reply or periodically checks for a reply. After it receives the reply, it continues. Servers wait for a request, process the request, and reply to the requestor.

Nothing in this pattern prevents one-way communication, in which a peer sends a request to a client but doesn't wait for a reply.

Connecting Client and Server through a Dispatcher

If the peers in your application system know one another's locations, the Forwarder-Receiver pattern, described in the preceding section, is useful. Sometimes, though, it's best not to have direct connections between clients and servers. In such a case, you can add a layer of indirection between clients and servers to hide the locations and make the locations easier to change. This intermediary is a *dispatcher.*

Answering telephone calls

When you call a company to buy something, you may get a receptionist. After you tell the receptionist who you want to speak to, he puts you on hold and connects you to the right salesperson. You don't need the direct contact information for the person you're calling because there is a receptionist to connect your call. You don't know how the receptionist lets the person know that she has a call, and you don't need to know.

You're the client in this example, and the server is the person that you want to reach. The receptionist is the dispatcher, who serves an intermediary role and hides the details of your server's location. If you're trying to buy something from a salesperson (the server) and a new salesperson has been assigned for your region, the receptionist can connect you to the correct, new salesperson without your having to know anything about the selling company's internal structure.

Issuing directions from a dispatcher

The dispatcher provides a name service, allowing servers to be given a name that the server can use instead of a physical location. It provides location transparency and simplifies maintenance if the server locations are variable. The name service can be a static repository, or the dispatcher can allow services to register and unregister during execution.

The dispatcher's name service makes it easy for you to add new servers to the system, too. You can also add duplicate servers to provide redundancy and higher availability of services.

The communication channel between the client and server is also created by the dispatcher. Figure 21-3 shows a message-sequence diagram for a system with a dispatcher. Unlike in the Forwarder-Receiver pattern, in which the client and server know where to find their peer, the dispatcher provides the initial address and then identifies the communication channel they'll use. After that channel is created by the dispatcher or the client (depending on the type of channel used), the client and server talk directly, without involving the dispatcher in the communications.

By now, you may be thinking that this pattern is similar to the Broker pattern in Chapter 12 — and you'd be right. Client-Dispatcher-Server is like a lightweight implementation of a direct communications broker system from the sidebar "Going for broker variations."

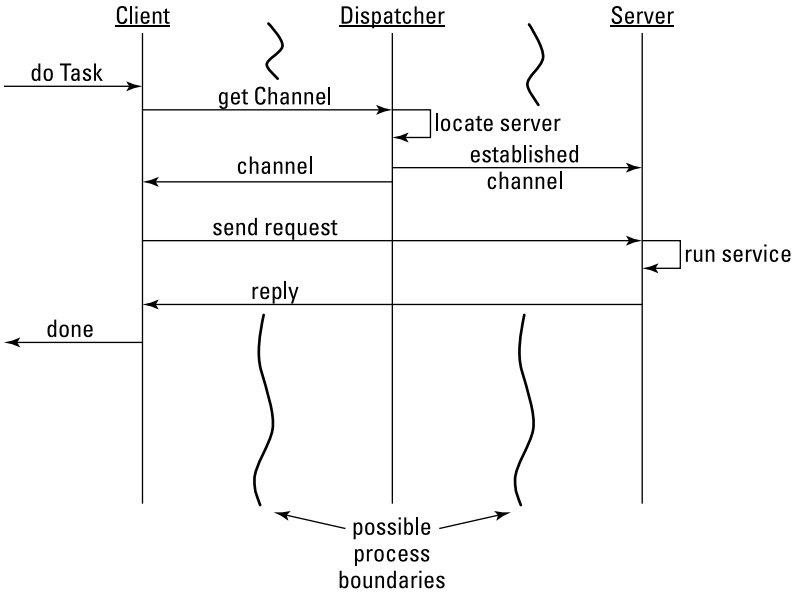


Figure 21-3: A Client-Dispatcher-Server pattern communication flow.

Implementing Client-Dispatcher-Server

Six steps are involved in designing a Client-Dispatcher-Server system. Some of these steps are related, so you don't need to do them strictly in the order presented here.

Step 1: Separate your application into clients and servers

In this step, you identify what parts of the system will function as servers and which will function as clients.

The same component can serve as both a client (requesting services) and a server (serving requests from clients). Also, the role that a component fulfills may change during execution.

Step 2: Determine how your components talk

This step focuses on the forwarder-to-receiver communication path and specifies how communications will occur. Your choices will be guided by what's available in your operating system. Different client-server pairs can use different communication methods.

If efficiency is important, you can use a low-level mechanism like TCP/IP. It's efficient and flexible, but programming it can be difficult. If your application needs to be portable, consider a mechanism like sockets. Sockets are available for most operating systems now, are sufficiently efficient for most applications, and aren't quite as closely tied to operating-system functions as they'd be if you programmed TCP/IP directly.

Shared memory is a fast alternative way for clients and servers to communicate if the clients and servers are on the same machine. If the client and server are located within the same address space, you can even use direct procedure calls between them.

Step 3: Define the component interaction protocols

In this step, you decide on the protocol that will be used to allow communications among the different pairs of entities: client-dispatcher, dispatcher-server, and client-server.

The messages between the client and server need to be in a format that both client and server recognize. In this step, you define the detailed structure of the messages that will flow between them.

Your protocols must define appropriate behavior when communications time out — that is, when replies aren't received within the required period. You also must consider the behavior of the protocol when communication fails. Alternative actions include retransmitting messages and reporting exceptions.

Step 4: Decide on server naming

The server name hides the server's physical location from the client. Only the dispatcher knows the name to location mapping. Server naming can be flexible, in the same way that Forwarder-Receiver naming is flexible. The names that you use shouldn't encode any location information. You can choose fixed constants or service descriptions for names. For more tips on naming, refer to Step 1 of "Implementing Forwarder-Receiver," earlier in this chapter.

Don't use an Internet IP address as your naming scheme. The IP address identifies a physical location, and the whole point of adding the dispatcher to your application is to provide location independence.

IP addresses were okay as names in Forwarder-Receiver earlier in this chapter because, in that pattern, the communications path is directly between the peers, whereas a dispatcher is intended to hide the physical locations.

Step 5: Build the dispatcher

In this step, you build the dispatcher component that will respond to the protocol you defined in Steps 3 and 4. You also establish communication between the client and the server, considering how the protocol and message sequence map to the communication mechanisms available to the system.

Some of the communication mechanisms may be limited resources. The number of sockets that a system can support simultaneously may be limited, for example. The dispatcher has to manage the connections and may have to place the request in an internal queue for the resource or even refuse a client request if it can't set up the required communications.

You must define and flesh out the protocol associated with dispatcher communications. Also make sure that you cover the error cases, such as when the server is unknown or unavailable or when a communication resource is unavailable.

One of the dispatcher's primary roles is to provide a name-mapping service that takes the server name known to the client and turns it into a physical address. The mapping can be statically defined before execution, or it can be dynamic. If the mapping is dynamic, it can change during execution, and the dispatcher must be able to process registration messages from servers by adding an entry in the name service directory (or by updating the record, if the server is already defined).

The performance of a single dispatcher in a system can become a performance bottleneck because it's a single component that processes all channel setup requests. You can use multithreading by providing a pool of threads in the dispatcher to improve response and execution times. One thread will complete a whole exchange on behalf of one client and server. If you have multiple threads, however, multiple clients can set up connections at the same time.

Step 6: Build the client and server components

Using what you designed and decided in the preceding steps, in this step you design and build your client and server components.

Publishing State Changes to Subscribers

One problem that arises in distributed systems is keeping the different elements consistent. When a change occurs in one element, the information must be conveyed to other elements in the system. This capability is

required by several architectures that I discuss in this book: Layers (see Chapter 9), Model-View-Controller (see Chapter 13), and Presentation-Abstraction-Control (see Chapter 14).

The Publisher-Subscriber pattern is very similar to the Observer pattern. Figure 21-4 shows the class diagram for Publisher-Subscriber. The interface between the publisher and subscriber can be any IPC mechanisms of your choice, including through the use of messaging-oriented middleware.

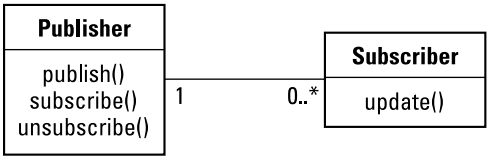


Figure 21-4: Publisher-Subscriber class diagram.

What I'm going to tell you about in this section is a variant on the Observer pattern. Besides changing the class names from ConcreteSubject to Publisher and ConcreteObserver to Subscriber, the biggest difference is that I won't tell you to use an abstract class. This pattern doesn't include the abstract classes, because although abstract classes help decouple the actual publisher and subscriber, they make the resulting code more complex.

This pattern has only two classes: *Publisher,* which sends notifications, and *Subscriber,* which receives the notifications. Subscribers register their interest in receiving these notifications, and publishers maintain a registry of its subscribers.

Implementing a Publisher-Subscriber mechanism involves the following three steps. You'll spend most of your effort designing the publisher component.

Posting a notice

You can post a notice on a bulletin board to publish information to others. The readers of the bulletin board are subscribers to the information. You, the publisher, don't need to know who all the readers are, and they don't have to hunt you down to find out what you have to say — they know to check the bulletin board.

Step 1: Define the publication policies

A publisher can publish in many ways. It can publish every notification event to every subscriber, or it can send some notifications to some subscribers and others to other subscribers.

In this step, you need to decide what internal events the publisher will publish to the subscribers. The publisher can push all event changes to the subscribers, or it can notify them about only the changes that they expressed interest in through the subscription process. If you make this choice, you need to include detailed subscription information in the subscriber registry.

In addition to providing selective notification (pushing notifications for only some events), the publisher can send complete updates — all the information about the notification — or a simple notice that some update has occurred. In this latter case, it becomes the subscriber's responsibility to request additional information from the publisher if the subscriber is interested.

Besides deciding what and how much to publish, you must decide how your publisher knows that something is available to be published. It can publish every update, or it can collect updates until a threshold is met and then publish.

Step 2: Define the publisher's interface

The next step defines the publisher's interface, which needs to include at least the following methods:

- **Subscribe:** The `Subscribe` method allows a potential subscriber to subscribe for update notifications.
- **Unsubscribe:** The `Unsubscribe` method allows a subscriber to stop receiving updates.

Step 3: Design the subscriber interface

The subscriber subscribes with the publisher for the information that it wants to receive. In Step 1, you designed the policies that define what the subscribers can ask for.

The primary part of the subscriber that needs to be designed is the `Update` method, which the publisher accesses when it sends its publication notifications. The `Update` method must receive the information from the publisher and ensure that it gets the information to the correct element within the subscriber component.

Chapter 22

Counting the Number of References

In This Chapter

- Using idioms as coding standards
- Counting references to dynamically allocated objects

This chapter contains something different from all the other chapters in this book: an idiom. In Chapter 6, I describe three categories of patterns based on the scope of the problems they tackle:

- **Architectural patterns** help you structure your whole solution architecture. I introduce some architectural patterns in Chapters 9 through 16.
- **Design patterns** help you solve individual design problems at the level of an object, component, or module. Chapters 17 through 22 contain design patterns.
- **Idioms** are language-specific patterns that help you overcome shortcomings in programming languages.

An idiom gives you insight into how to overcome limitations in a particular programming language, essentially telling you "tricks" for making the most of a language. It also can help you understand how to use the language and read programs written in it.

This chapter describes one idiom that I've found to be useful in many situations: Counted Pointer. This idiom helps you program in the C++ language, but the concept is applicable in other situations, too.

Problem: Using the Last of Something

Sometimes, your C++ program has data objects that are too big to copy efficiently. You can use multiple pointers to refer to one copy of the big data structure to provide some flexibility when this is the case. In a similar

situation, your program has multiple pointers to a specific resource or bit of information that you can't copy because it absolutely must be the only copy in the system. This may be the case when there's a central constant or central object that every other part of the system should refer to.

The system can delete the item and release the resources only when the last pointer is done using it. How do you know, however, when there are no more active pointers to the resource or structure? Following are three ways to try to find out.

First try: Passing objects with pointers

Passing objects as parameters to functions is common in object-oriented C++ programs. Pointers usually accomplish this task. Without care, however, the situations shown in Figure 22-1 can occur:

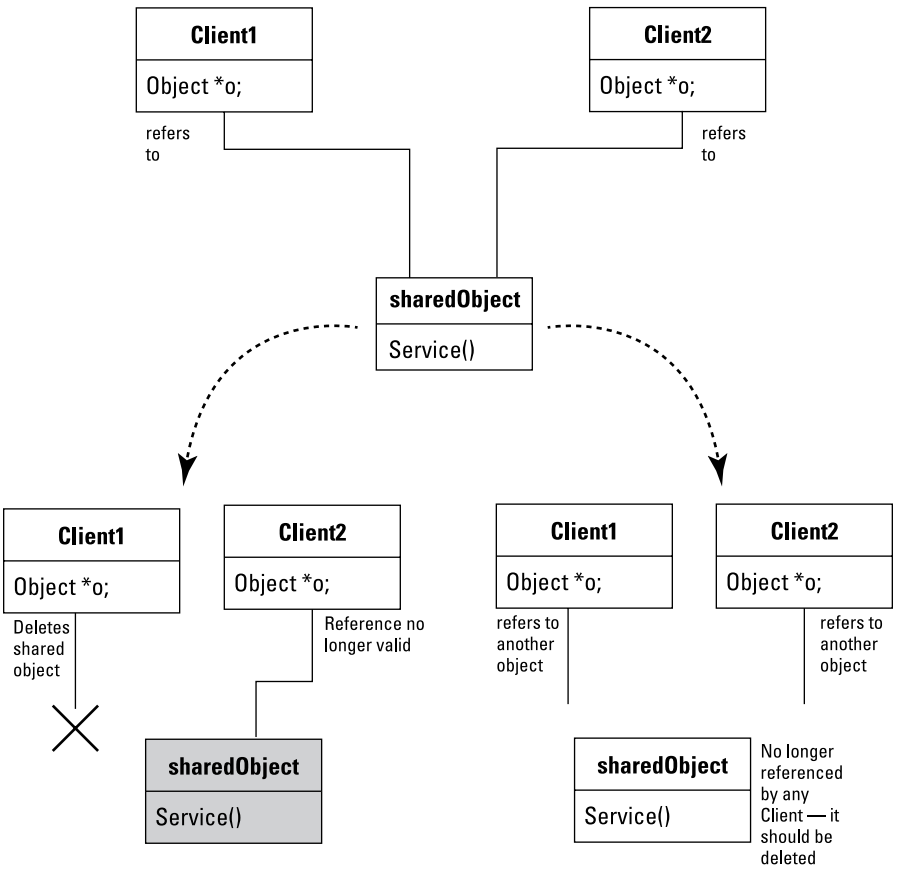


Figure 22-1: What can go wrong with pointers to shared objects.

- One client of an object deletes the shared object out from under another reference.
- All the clients stop using the shared object, but none of them bothers to delete it — thereby leaving it to waste memory.

Second try: Passing objects by copying

You can solve this problem by avoiding pointers and passing the objects by value — where the actual value is copied and passed. The C++ compiler will delete the copied value automatically when it goes out of scope.

This solution doesn't work in all situations, however. If the object being passed is large, both the execution time required to copy the object and the amount of duplicated memory are large — representing two strikes against the copying. If your application is creating dynamic structures of objects such as trees or directed graphs, passing objects around by value will be impossible.

Another situation in which copying is a poor choice is when you deliberately want to pass around references to the same item instead of copying it — such as when you want to point to a shared item so that it can be updated in one place and make the updates available elsewhere immediately and transparently.

Third try: Using the Counted Pointer idiom

The following forces make this problem hard to solve and explain why the preceding answers don't work:

- Sometimes, it's inappropriate for a class to pass objects by value.
- The same object may need to be shared by several clients.
- You don't want to have *dangling references* (references to objects that have been deleted).
- Shared objects should be deleted when they're no longer needed.
- The solution shouldn't require too much extra code.

In this case, consider a third solution: using a counted pointer to release resources. I show you how in the following section.

Solution: Releasing Resources with the Counted Pointer Idiom

The Counted Pointer idiom addresses the forces that I mention in the preceding section and helps you solve the problem of memory management with dynamically allocated objects. The solution involves making only a small addition to the object being shared and adding another class.

Two classes are involved in the solution:

- **Body:** The *body* is the object that will be referenced and shared; it probably exists already. A reference counter in the body keeps track of the number of pointers to it by other objects.
- **Handle:** The *handle* is introduced as the only class in the system that's allowed to have a reference directly to the body.

All references to the body are made through the handle. Accesses to the body through the handle manipulate the reference counter, incrementing it when a new class points to the handle and decrementing it when a reference to the handle is eliminated. Figure 22-2 shows the structure of the solution.

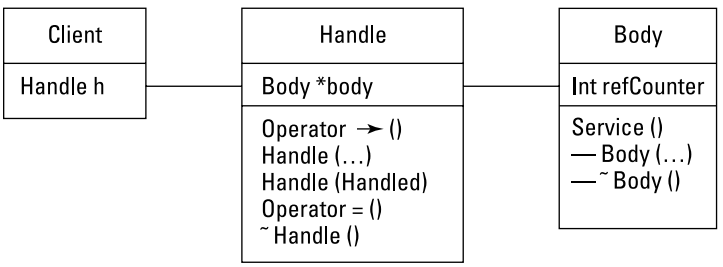


Figure 22-2: A class diagram of Counted Pointer.

The handle objects are passed by value throughout the system, which causes them to be allocated and destroyed automatically. If the reference count was stored in the handle, it would be copied, which would result in handles racing to delete or preserve the body. The actual reference count is stored in the body to prevent this problem with the counter being duplicated.

Figure 22-3 shows a snapshot of the system with several handles all pointing to the same body.

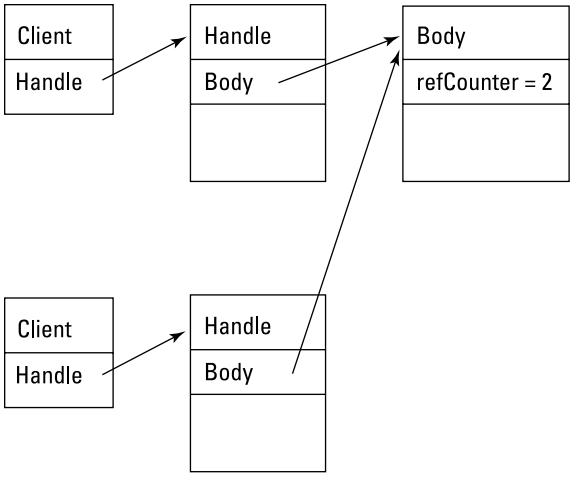


Figure 22-3: Handles and a body in midexecution.

Implementing Counted Pointer

Implementing Counted Pointer takes just a few steps:

1. **To keep the body safe from direct accesses, make its constructors and destructors private.**
2. **Make the `Handle` class a friend of the `Body` class.**

   This step gives the `Handle` class access to the internals of the `Body` class and improves efficiency.

   This exception for efficiency is noted in the Necessary Friends coding standard (see the nearby sidebar "Styling your code with idioms").

3. **Add a reference counter to the `Body` class.**

   Because the handle is a friend of the body (refer to Step 2), you don't need to create setters and getters for the reference count.

4. **To the `Handle` class, add a data member that points to the `Body` class.**
5. **In the `Handle` class, implement a copy constructor and an assignment operator that copy the pointer to the body object and increment the body object's reference count.**
6. **Implement the `Handle`-class destructor that decrements the body's reference count.**

If the count reaches `0`, the `Handle`'s destructor also deletes the body object.

7. **Implement the arrow operator as follows, making it a public member function:**

```
Body* operator->(){ return body; }
```

8. **Extend the `Handle` class with constructors that create the body for the first time.**

   These constructors should set the reference counter to `1`.

The C++ code shown in Listing 22-1, which comes from *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley), shows a sample implementation of the steps.

Although the following Counted Pointer solution is a C++ solution, it's applicable to much more than just C++.

Listing 22-1: Counted Pointer in C++

```
class Body {
public:
// methods providing the body's functionality
// to the world
  void service();
  // further functionality ...
private:
  friend class Handle;
  // parameters of constructor as required
  Body(/*...*/) { /* ... */ }
  ~Body() { /* ... */ }
  int refCounter;
  char BigData(/* ... */);
};

class Handle {
public:
  // use Body's constructor parameters
  Handle(/* ... */) {
     body = new Body(/* ... */);
     body->refCounter = 1;
  }
  Handle(const Handle &h) {
     body = h.body;
     body->refCounter++;
  }
  Handle & operator=(const Handle &h) {
     h.body->refCounter++;
     if (--body->refCounter) <= 0)
```

```
        delete body;
     body = h.body;
     return *this;
  }
  ~Handle() {
     if (--body->refCounter <= 0)
        delete body;
  }
  Body* operator->() { return body; }
private:
  Body *body;
};

// example use of handles ...
Handle h(/* some parameter */);
//create a handle and also a new body instance
{
  Handle g(h); // create just a new handle
  h->service(); g->service();
} // g goes out of scope and is automatically deleted

h->service(); // still possible
// after h goes out of scope the body instance is
// automatically deleted.
```

Styling your code with idioms

Idioms are useful for explaining to other members of your team, project, and company how to write software. For one thing, the idiom's name enters people's vocabularies easily, so when someone says "Counted Pointer," for example, everyone knows what she means.

You've probably seen how easy it is to read and understand a program written in a consistent style. Coding standards help everyone on a project know the normal way of writing code. The standards range from guidelines about the number of spaces to the correct format for variable names to the directory where source files are stored — and everything in between.

Here's an example idiom, Necessary Friends, that's used as a coding standard:

- **Context:** C++ provides the capability for one class to friend another, which allows the class to look into and access private methods and data in the friended class.
- **Problem:** Friending violates encapsulation and information hiding, so you generally should avoid it. To achieve needed efficiency, however, one class sometimes must access the internals of another.
- **Solution (the coding standard):** Avoid establishing friends between classes. Use this capability only in extreme situations, when efficiency requires it.

Version 11 of C++, first approved in 2011, provides an even easier way to implement the basic Counted Pointer idiom. This version of the C++ language standard introduces the `shared_ptr` class, with all the parts that I just described:

```
template< class T > class shared_ptr;
```

For more information, visit this website: `http://en.cppreference.com/w/cpp/memory/shared_ptr`.

Seeing some Counted Pointer variations

The preceding section explains how to create a Counted Pointer structure that separates the handle, which is passed around, and a body with a reference count. Several variants may work better for you in some situations, however. This section provides short descriptions and pointers to more information.

Coplien's Counted Body idiom

The Counted Body (or Reference Counting Idiom) variant helps improve performance when the body objects are large. Each client thinks that it's using its own body object, but that object is shared with other clients, as described in the preceding discussions of the Counted Pointer idiom. When an action that will change the body object is about to happen, a copy of the body is made, and the reference count of the original body is decremented. This variant idiom keeps the bodies the same for as long as possible but creates a new copy that can be changed when necessary.

This variant is introduced by James Coplien in *Pattern Languages of Program Design 4,* by Brian Foote, Neil Harrison, and Hans Rohnert (Addison-Wesley).

Coplien's Detached Counted Body idiom

Another variant, also by Coplien, wraps existing classes with a class that contains the reference counter. This variant adds another class between the handle and the body that contains the reference count. It continues the pass-through that you implemented in the handle of body-specific methods and requests. The extra class adds to the execution-time expense of this variant. The benefit is that the `Body` class needs no modification at all to add the counter.

This variant, shown in Figure 22-4, is described in *Pattern Languages of Program Design 4* (see the preceding section).

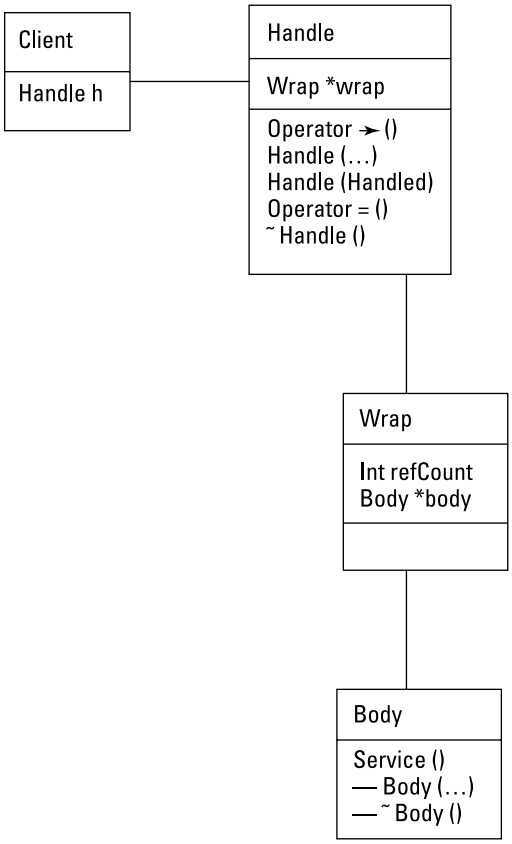


Figure 22-4: Wrapping the body with a counter class.

Koenig's Detached Counted Body idiom

Andrew Koenig suggests another variant in "Another Handle Variation," which appeared in the *Journal of Object-Oriented Programming,* Vol. 8, No. 7 (1995). In this variant, the counter is the `Count` class (see Figure 22-5). This class isn't *between* the `Handle` and `Body` classes; it's *parallel* to the `Body` class. The `Count` class is pointed to and used by the `Handle` to maintain the

reference counts. The `Count` class can maintain reference counts for several `Body` classes by adding more `refCount` attributes. The `Handle` class grows because it needs extra space to manipulate two classes: `Count` and `Body`.

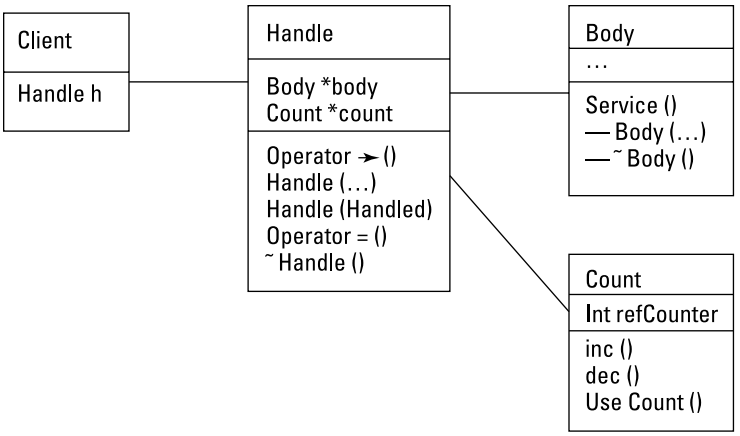
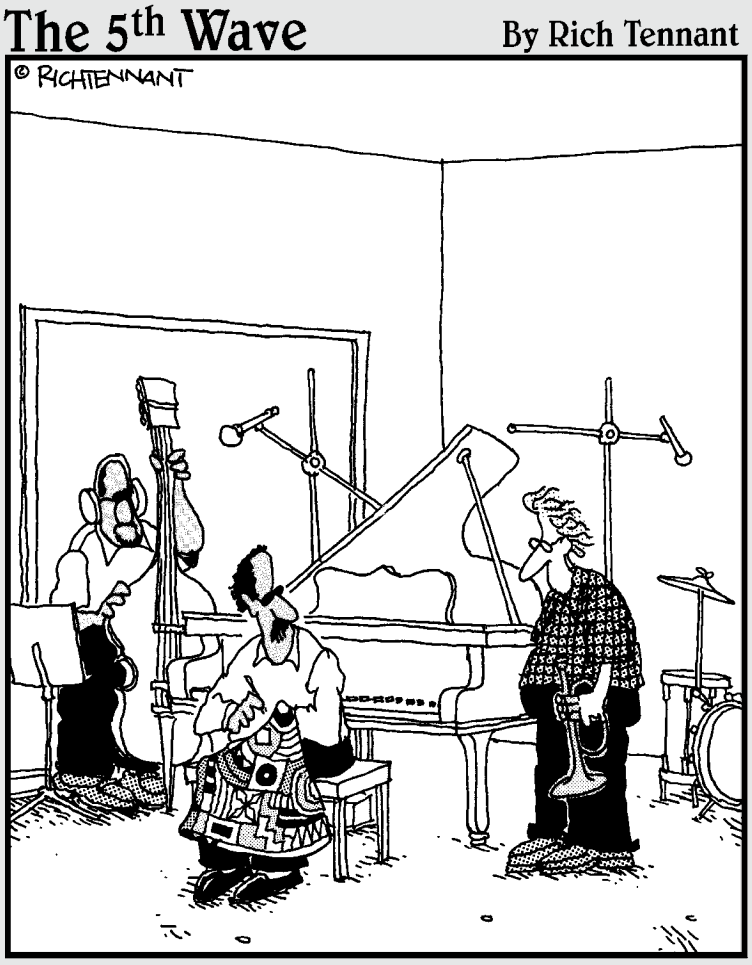


Figure 22-5: Koenig's variant with a `Count` class.

Part V
The Part of Tens



"I don't work with a pattern, man. I just make it up as I go along."

In this part . . .

The Part of Tens is where I share some helpful tips on using patterns in your own work. I list ten patterns that every developer should know, as well as ten resources that you can seek out to continue your research on patterns. Finally, I provide ten ways to get more deeply involved in using, writing, and promoting software patterns.

Chapter 23

Ten Patterns You Should Know

In This Chapter

- Introducing new behavior and ideas
- Checking and patching your software
- Keeping track of events
- Staying signed in

In this book, I've told you about 17 specific software patterns that you'll find useful as you design and build software systems. They represent just the tip of the iceberg, however, because not all patterns are about software. Many other patterns are available, covering a wide range of problem categories, including people, organizations, and buildings. This chapter lists ten more patterns that you should know.

My goal in giving you this list is to fuel your interest in patterns by showing you that patterns cover a wide range of subject areas. Even if you don't use any of the architectural patterns in Part III again, you'll still find some patterns useful.

Patterns exist in a context, and the most useful patterns are part of a pattern language (see Chapter 6). The patterns in this chapter serve as an introduction to the pattern languages or collections that they're part of.

Special Case

The Special Case pattern describes how to have your system seamlessly provide alternative, special-case behavior. It's a variant and refinement of the pattern Null Object, for which many authors have written many variants; it's also a refinement of the Strategy pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).

When you're designing with objects, you define the classes and objects to handle some responsibility or data. Sometimes, though, you need to remember that there isn't anything to save or to do, which is a special case of the data and responsibilities. Figure 23-1 shows a simple class diagram of Special Case.

The Special Case pattern is from *Patterns of Enterprise Application Architecture,* by Martin Fowler (Addison-Wesley).

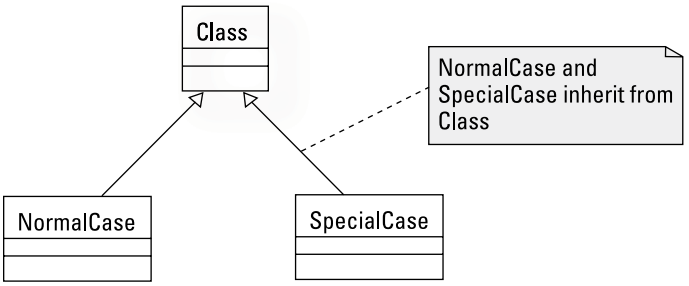


Figure 23-1: The classes of Special Case.

Do Food

The Do Food pattern is part of a collection of patterns for introducing new ideas into an organization. You've probably seen the pattern in your own life: When you want to get a group of people together for some purpose, you "do food." You can have people bring their own, or you can provide the food, but the basic point is the same: The way to recruit a group of people is to make food available at the meeting.

Do Food is from *Fearless Change: Patterns for Introducing New Ideas,* by Mary Lynn Manns and Linda Rising (Addison-Wesley).

Leaky Bucket Counter

This design pattern is a cornerstone of fault-tolerant programming and is also useful for resource-allocation issues. It tells you how to keep track of events and to take action only when the frequency of the events is larger than you're willing to allow.

Figure 23-2 shows the general principles of Leaky Bucket Counter. Error reports fill the bucket, while an allowable number of errors is deducted from the bucket constantly as it leaks. If the rate at which the bucket fills is larger than the leaking rate, an error is triggered.

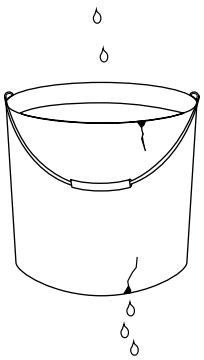


Figure 23-2: Principles of Leaky Bucket Counter.

Leaky Bucket Counter is from my book *Patterns for Fault Tolerant Software* (Wiley).

Release Line

After you release a version of your software to your users, you'll probably have to fix some problems. You can expect that your users will update to the latest version to get the fixes and other enhancements, but frequently, they won't want to update or can't take the upgrade when you want them to. The Release Line pattern describes a way to create a release line when you ship your software. This line will be maintained and patched in parallel to the main software development. Patches are applied to the release line and given to customers. Periodically, the patches are brought back into the main development product, as shown in Figure 23-3.

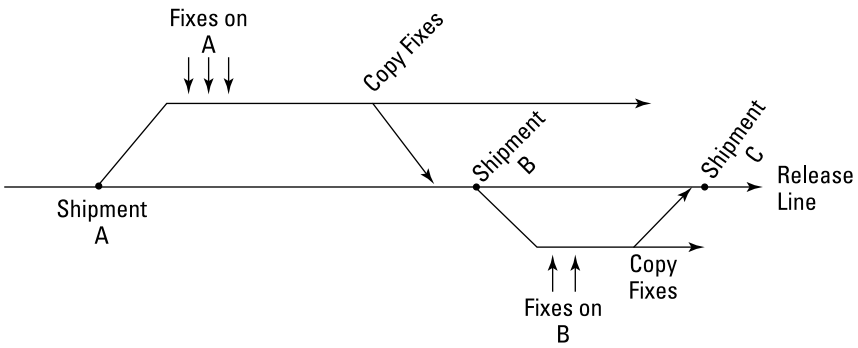


Figure 23-3: The Release Line pattern.

Release Line is from *Software Configuration Management Patterns: Effective Teamwork, Practical Integration,* by Stephen P. Berczuk and Brad Appleton (Addison-Wesley).

Light on Two Sides of Every Room

Think about the place in your home where you most like to sit. Where is the light coming from? Odds are, it's coming from two sides of the room. You can reproduce that effect in your own architecture, thanks to the Light on Two Sides of Every Room pattern.

The easiest way to think about this pattern is to imagine a corner room with windows on two sides, but the light can come from a variety of sources, such as windows, skylights, and open doors.

This pattern is from the collection of building patterns in *A Pattern Language: Towns, Buildings, Construction,* by Christopher Alexander, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel (Oxford University Press).

Streamline Repetition

Streamline Repetition gives the user an easy way to repeat things that need to be repeated. Examples include the find and replace box in many editors, the ability to record sequences of events as macros that can be repeated with a single command, shell scripting, and user-defined shortcuts.

The Streamline Repetition pattern is from *Designing Interfaces,* 2nd Edition, by Jenifer Tidwell (O'Reilly). The book talks about all aspects of design for user interfaces, but this pattern applies to most any user interface problem.

Observer

Many developers view the Observer pattern from *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional) as the most important pattern in that book. Others think that you need to know all 23 patterns.

The Observer pattern sets up a one-to-many relationship between a subject and an observer. When the singular object changes state, all the other "observers" are notified. This pattern is good for supporting broadcast communications from one-to-many, and it's particularly useful in these situations:

- When an abstraction has two different aspects, and one of those aspects depends on the other one
- When the number of objects that are observing the state change is unknown
- When the object being observed shouldn't know anything about the objects that are observing it

The class's arrangement is shown in Figure 23-4.

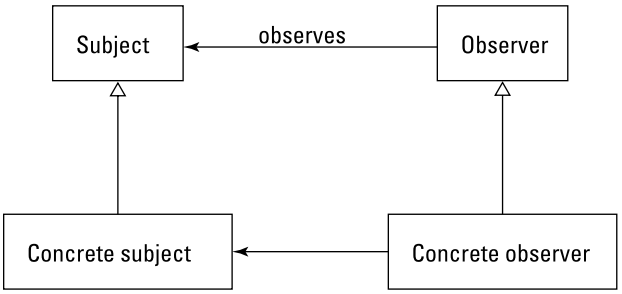


Figure 23-4: The Observer pattern.

Sign-In Continuity

One aspect of social-networking websites that I hate is links that don't take me where I want to go. This happens when I haven't logged in yet, so I get taken to the login page, and then the website decides that I should go to my home page. That page isn't where I wanted to go, however, and by now, I can't remember what I wanted to look at anymore. The Sign-In Continuity pattern addresses this situation and recommends that the software return the user to the context he or she was in before being asked to sign in.

Sign-In Continuity is from *Designing Social Interfaces: Principles, Patterns, and Practices for Improving the User Experience,* by Christian Crumlish and Erin Malone (O'Reilly).

Architect Also Implement

This pattern deals with organizational issues. System architects can become isolated from real system construction. When this happens, they begin to make architectural decisions that don't fit with the existing structure of the system or even with what is technically feasible. Another problem is that the people who actually build the system may not understand the architect's vision of it.

One way to prevent this problem is to use the Architect Also Implement pattern, which is part of a language related to the organizational aspects of software development. You can find it in *Organizational Patterns of Agile Software Development,* by James O. Coplien and Neil B. Harrison (Prentice Hall).

The CHECKS Pattern Language of Information Integrity

In 1994, Ward Cunningham brought his CHECKS Pattern Language of Information Integrity to the first software-patterns conference — Pattern Languages of Programming, or PLoP (see Chapter 5). This language contains ten patterns for telling good input from bad. It also describes recording the fact that the input was bad and continuing processing even with the bad input. The methods are designed to make the checks without overly complicating your program or making them inflexible for future changes.

The ten patterns included in the language are

- Whole Value
- Exceptional Value
- Meaningless Behavior
- Echo Back
- Visible Implication
- Deferred Validation
- Instant Projection
- Hypothetical Publication
- Forecast Confirmation
- Diagnostic Query

CHECKS appears in *Pattern Languages of Program Design,* by James O. Coplien and Douglas C. Schmidt (Addison-Wesley). A version of CHECKS is available on the web at `www.c2.com/ppr/checks.html`.

I include a language of ten patterns here because CHECKS is an early example of a small number of patterns that work together to solve a problem powerfully. Better yet, it's still useful!

Chapter 24

Ten Places to Look for Patterns

In This Chapter

- Checking out language collections
- Looking at user interfaces
- Investigating apprenticeships

Chapter 23 presents ten patterns that you should know. In this chapter, I tell you about ten sources of those collections. The books and websites listed here are ones that I turn to frequently or that many other people in the pattern community use regularly. These pattern collections cover a wide range of problem categories, so no matter what your interest, you should find something interesting. You can find pointers to even more patterns at `www.hillside.net`.

Because patterns can be about more than just software, the first pattern source that I mention here is about buildings.

A Pattern Language

A Pattern Language: Towns, Buildings, Construction, by Christopher Alexander, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel (Oxford University Press), is one of the works that got software people interested in patterns in the first place. This very readable book contains 253 patterns, ranging from designing a nation-state to decorating your home with the things you like and the things that tell the visitor who you are (or that tell the visitor your story). If you're interested in reading very well-written patterns — or are in the process of redesigning your home — you should check it out.

Pattern-Oriented Software Architecture

In this book, I focus on the patterns in the first volume of the series, *Pattern-Oriented Software Architecture: A System of Patterns,* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley). The series, however, has four more volumes:

- ***Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley):** This book contains patterns related to service access and configuration, event handling, synchronization, and concurrency.
- ***Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management,* by Michael Kircher and Prashant Jain (Wiley):** Resource acquisition, life cycle, and release patterns are presented in this book.
- ***Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing,* by Frank Buschmann, Kevlin Henney, and Douglas Schmidt (Wiley):** This book adds to the patterns that I discuss in Parts III and IV of this book, providing patterns related to the technical aspects of distributed systems. The patterns in this volume create a pattern language.
- ***Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages,* by Frank Buschmann, Kevlin Henney, and Douglas Schmidt (Wiley):** This volume, which wraps up the series, is an excellent source of detailed information about patterns, expanding on what you find in Part II of this book.

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional), contains 23 patterns that explain how to solve many problems associated with object-oriented programming. The patterns are divided into three categories:

- **Structural:** Patterns that help you put together the right arrangement of patterns to solve the problems
- **Behavioral:** Patterns that help you spread responsibility among your objects
- **Creational:** Patterns that help your system create new objects

This book is the first exposure many people get to patterns and a book that anyone doing object-oriented design should know. You've seen it referenced throughout this book.

Domain-Driven Design

Domain-Driven Design: Tackling Complexity in the Heart of Software, by Eric Evans (Addison-Wesley), is considered one of the most influential software-pattern books. When you start reading it, however, you'll notice that the patterns aren't thrust in your face. The book contains patterns, but they serve the purpose of explaining Evans's design technique rather than being presented as patterns by themselves.

Pattern Languages of Program Design

The five volumes of the *Pattern Language of Program Design* series contain the best of the patterns presented at the early Pattern Languages of Programming (PLoP) conferences. (See Chapter 5 for more information on these conferences.) The books contain an eclectic mix of patterns covering a wide range of topics, from the practice of patterns to organizational patterns to specific problem categories.

- *Pattern Language of Program Design,* Volume 1, edited by James O. Coplien and Douglas C. Schmidt (Addison-Wesley), contains the patterns from the first PLoP conference, held in 1994. The patterns range from very complete languages like CHECKS (see Chapter 23) to chapters that contain the author's thoughts about patterns rather than actual patterns.
- *Pattern Language of Program Design,* Volume 2, edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth (Addison-Wesley), contains patterns that were reviewed and discussed at the 1995 PLoP conference. It presents no discussions of patterns. Early versions of Leaky Bucket Counters and Architect Also Implements (see Chapter 23) are in this volume.
- Starting with *Pattern Language of Program Design,* Volume 3, edited by Robert Martin, Dirk Riehle, and Frank Buschmann (Addison-Wesley), the books contain patterns from several PLoP conferences, including EuroPLoP. The patterns also span several years. In other words, the books got more selective, so only the best patterns appear.

- *Pattern Language of Program Design,* Volume 4, edited by Neil Harrison, Brian Foote, and Hans Rohnert (Addison-Wesley), continues the documenting of patterns from both PLoP and EuroPLoP. Some patterns that I wrote for managing input and output in embedded systems appear in this volume.
- The series stops today with *Pattern Language of Program Design,* Volume 5, edited by Dragos Manolescu, Markus Voelter, and James Noble (Addison-Wesley), which continues to bring together patterns for a variety of computing domains in one place.

Patterns for Time-Triggered Embedded Systems

Throughout this book, I've been telling you that patterns aren't exclusively about objects, and this selection of patterns takes you a long way from objects.

Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers, by Michael Pont (Addison-Wesley), contains patterns to help software engineers with experience in desktop systems expand their knowledge into the world of embedded systems. It also helps hardware engineers understand the software that goes into embedded systems and enables students to combine hardware and software to make something.

This massive, comprehensive book is full of tips and techniques for getting the most out of almost any microcontroller, including code and circuit diagrams.

It's also available on the author's website: `www.tte-systems.com/books/pttes`.

Software Configuration Management Patterns

Stephen P. Berczuk and Brad Appleton created *Software Configuration Management Patterns: Effective Teamwork, Practical Integration* (Addison-Wesley) to describe effective ways of managing projects from the software-configuration-management perspective. The pattern Release Line, mentioned in Chapter 23, comes from this book.

Patterns of Enterprise Application Architecture

In the first part of *Patterns of Enterprise Application Architecture* (Addison-Wesley), Martin Fowler introduces several issues and concerns that surface during the design of enterprise architectures. Fowler defines *enterprise applications* as those that deal with storing and processing large amounts of data and applications that use that data to support business processes. In the second part of the book, Fowler introduces patterns that solve those problems.

Welie.com

The patterns in Parts III and IV of this book deal with software, but a wealth of patterns for user interfaces is available too. Both Sign-In Continuity and Streamlined Repetition, mentioned in Chapter 23, manage user interaction with systems.

The user-interface community has an excellent resource in a website maintained by Martijn van Welie, `www.welie.com`. This site hosts a large collection of user-interface patterns and has links to many other collections. It's an excellent entry point into the world of user-interface patterns.

Apprenticeship Patterns

Apprenticeship Patterns, by Dave Hoover and Adewale Oshineye (O'Reilly), is one of my favorite recent books of patterns, capturing patterns related to being a good apprentice. The book targets readers who are interested in becoming better, more-refined software practitioners. It provides tools and tips (including patterns) for mastering the wealth of information available to software professionals and growing your career.

Chapter 25

Ten Ways to Get Involved with the Pattern Community

In This Chapter

- Writing — and writing about — patterns
- Mentoring newcomers
- Taking advantage of pattern-community resources
- Starting your own pattern-related groups

If you've arrived at this chapter after reading the rest of the book, you may be wondering where to go from here. This chapter gives you ten tips to help you get more out of patterns and contribute to the worldwide pattern community. Each tip is a step toward passing through the gate (refer to Chapter 8).

I used these techniques when I was just getting started with patterns. I hope that they'll help you as much as they helped me.

Advocate Using Patterns

Tell others about how wonderful patterns are. If you're reading this book, you probably think (as I do) that patterns can benefit anyone doing software. Start telling your colleagues and friends about how useful patterns have been for you.

Please don't hype patterns, though. I remember too well the days when we were told that "[pick your innovation] would save the world," only to later find that, although it was a useful tool, it wasn't the last word in software.

Write About Your Experiences Using Patterns

After successfully using patterns in a few projects, write down your experiences. The unique insights that you have about how to find patterns or include them in your software can help others.

Even though you may feel like you're still a rookie with patterns, there are many things you can tell others about, such as:

- How you got started with patterns
- How you found the patterns that you find useful
- What patterns you keep in your pattern catalog and why
- What "translations" you had to make to the patterns to fit your environment
- What difficulties you encountered using the patterns and how you overcame them
- What unique problem patterns helped you surmount in your projects

Compile a Catalog of Your Work

Chapter 7 gives you the details on writing a personal pattern catalog, which also can be a portfolio of your work. Write a pattern catalog for your workplace or software project. Make it something that you and your teammates can use over and over again.

Because patterns are useful for defining vocabulary, your catalog can be an introduction to the project, helping newcomers get a handle on the terminology and trade-offs in your domain.

Mentor Someone

If someone new starts looking into patterns, mentor him. Help him to build his own pattern catalogs. Teach him about the most important patterns that you know. Work with him to bring him to the same level of understanding about patterns that you have.

You don't need to know everything; you just have to stay a step ahead of your protégé. Mentoring is an opportunity for you to grow, too, because learning by teaching is a well-known pattern.

Help Index Patterns

No comprehensive catalog or index of all the available patterns exists, so you can pitch in to help the pattern community catalog patterns. Some people have created indices, but the efforts usually fade away after a while, so the pattern community always needs indexers. Within the user interface community Martijn van Welie's website (`www.welie.com`) has patterns and links to other patterns, letting it serve as an index. Either by yourself or with a group, look at the existing pattern literature, create references and cross-references (much as you do in your own pattern catalog; see Chapter 7), and make the information available to the rest of the pattern community. The Hillside Group (`www.hillside.net`) is the ideal center for any indexing effort.

Join a Mailing List

If you want to help other people review their patterns, discuss patterns in general, or get answers to questions about how to use them, mailing lists about patterns may be helpful.

The pattern mailing lists at the University of Illinois — where Ralph Johnson, one of the authors of *Design Patterns: Elements of Reusable Object-Oriented Software,* works — have been around the longest. Information about how to subscribe is available at `www.hillside.net/patterns/mailing-lists`. Here are some of the lists available at the U of I:

- patterns@cs.uiuc.edu is for presenting and describing software patterns.
- business-patterns@cs.uiuc.edu is for presenting and describing business patterns.
- patterns-discussion@cs.uiuc.edu is for discussion of patterns in general.
- gang-of-4-patterns@cs.uiuc.edu is about the design patterns in Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional).
- siemens-patterns@cs.uiuc.edu is about the patterns described by the Siemens guys. (Hey, wait a minute — this is a mailing list about the patterns in this book!) People use the list to talk about the finer points of, or ask questions about, the patterns in Pattern-Oriented Software Architecture: A System of Patterns, by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley).
- organization-patterns@cs.uiuc.edu is for discussing patterns involving organizations.

- corba-patterns@cs.uiuc.edu is about patterns described in CORBA Design Patterns, by Thomas Mowbray and Raphael Malveau (Wiley), and related patterns.
- antipatterns@cs.uiuc.edu concerns antipattern refactoring and the book AntiPatterns, by William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray (Wiley).
- scm-patterns@cs.uiuc.edu is about patterns for software configuration management.
- telecom-patterns@cs.uiuc.edu is about patterns for telecommunications.

Join a Reading Group

Reading groups look through patterns or other literature to learn as a group. There may be a reading group near where you live or work. Join it and start learning with others. You'll meet like-minded people who can also learn from you and your experiences.

You can find the patterns to study in such a group in many places. Patterns have appeared in many, many books and magazines over the past decades. The Hillside Group (`www.hillside.net`) can help with its listing of books.

Journals such as *IEEE Computer, IEEE Software,* and *The Communications of the ACM* all have had special issues devoted to patterns. There is also the peer-reviewed journal *LNCS Transactions on Pattern Languages of Programming;* more information is available at `www.springer.com/computer/lncs?SGWID=0-164-2-470309-0`. Any of these can provide your reading group articles to study.

If you can't find an existing group, you can start your own by using the suggestions at `www.industriallogic.com/papers/khdraft.pdf`.

Write Your Own Patterns

When you start writing your own patterns, think about what you're an expert in. To narrow down the area(s) in which you're an expert, keep some notes about the specific questions that people ask you over and over, such as the following:

- How do I get the compiler to work?
- How do I set up the IDE?
- How do I use the XYZ protocol?

If there isn't already a pattern that explains how you answer the questions — write your first pattern as a solution to one of those questions.

Patterns are proven solutions. If you don't know the solution to the problem, you don't have a pattern.

Think about your intended audience. Who do you want to read your pattern and benefit from it? If you're writing about the things that people ask you about, your audience is probably your co-workers or friends. Write the pattern to answer their questions.

When the pattern is completed, label it a candidate pattern, and have your colleagues read and review it (refer to Chapter 5). After a review or two, the pattern will be able to stand on its own, and people can use it instead of asking you the same questions over and over.

Your pattern may be immediately useful to your workgroup or circle of friends. If you want it to be more widely useful, you need to make it widely available. If you're confident in your pattern and your skills, you can just post it on your website and hope that people find it and use it. A better approach, however, is to have some of your fellow pattern users read it and offer suggestions for improving it and making it more widely known.

Chapter 4 contains a thorough discussion of the components of a pattern. Chapter 5 gives you an introduction to writing your own patterns and provides a template that you can fill in.

Attend a Pattern Conference

Several conferences exist to review or discuss patterns, including the Pattern Languages of Programming (PLoP) series (see Chapter 5), which is held in the United States every year. After a couple years of PLoP, a group from Europe created the EuroPLoP conference, which is held each year in Germany. Many other conferences have followed. These conferences are usually geared toward reviewing patterns to help make them better.

Reading a variety of patterns and participating in a discussion of how to make them better is a great way to become exposed to and to learn many different concepts. Information about these patterns and when they are next going to be held is available at `www.hillside.net/conferences`.

Start a Writers' Workshop

If your mentoring and advocating activities have been going well, you may be surrounded by others who also want to start writing patterns. Encourage them, share your experiences, and mentor them. Writers' workshops (see Chapter 5) are good venues for reviewing and improving patterns. If there's no group near you, start a writers' workshop of your own.

Many resources can help you get one going. A good starting point is `http://members.cox.net/rising11/Articles/WritersWorkshop.doc`. (***Note:*** This URL automatically downloads a Microsoft Word document to your computer.)

Index

Numerics

• A •

• B •

• C •

• D •

• E •

• F •

• G •

• H •

• I •

• J •

• K •

• L •

• M •

• N •

• O •

• R •

• S •

• T •

• U •

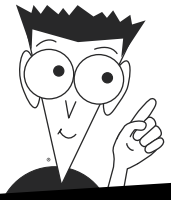
• V •

• W •

• X •

• Y •

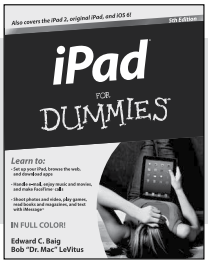
• Z •



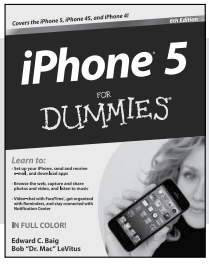
FOR DUMMIES®

Making Everything Easier!™

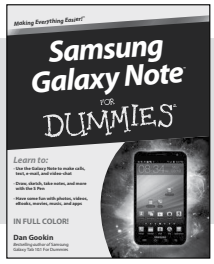
GADGETS



978-1-118-49823-1

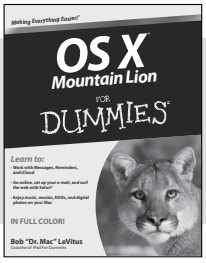


978-1-118-35201-4

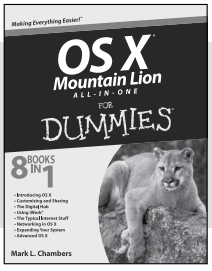


978-1-118-38846-4

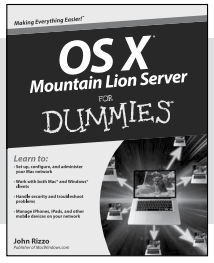
MAC OS X MOUNTAIN LION



978-1-118-39418-2

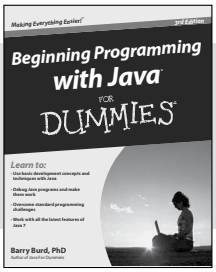


978-1-118-39416-8

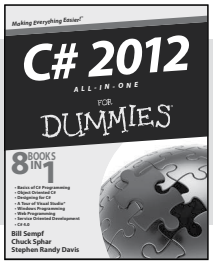


978-1-118-40829-2

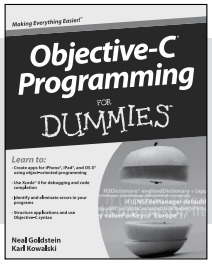
PROGRAMMING LANGUAGES



978-0-470-37174-9



978-1-118-38536-4



978-1-118-21398-8

Canon EOS Rebel T4i/650D For Dummies
978-1-118-33597-0

Digital SLR Cameras and Photography For Dummies, 4th Edition
978-1-118-16169-2

Digital SLR Photography For Dummies
978-1-118-45738-2

iMac For Dummies, 7th Edition
978-1-118-20271-5

iPad For Seniors For Dummies, 5th Edition
978-1-118-49708-1

iPhone 5 For Seniors For Dummies, 2nd Edition
978-1-118-37542-6

Kindle Fire HD For Dummies
978-1-118-42223-6

Nexus 7 For Dummies (Google Tablet)
978-1-118-50873-2

Nikon® D3100TM For Dummies
978-1-118-00472-2

Nikon D3200 For Dummies
978-1-118-44683-6

Nikon D5100 For Dummies
978-1-118-11819-1

NOOK eReaders For Dummies, Portable Edition
978-1-118-44044-5

Photoshop Elements 11 All-in-One For Dummies
978-1-118-40822-3

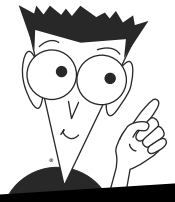
Photoshop Elements 11 For Dummies
978-1-118-40821-6

R For Dummies
978-1-119-96284-7

Sony Alpha SLT-A35 / A55 For Dummies
978-1-118-17684-9

Surface For Dummies
978-1-118-49634-3

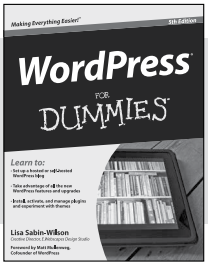
Available wherever books are sold. For more information or to order direct go to www.wiley.com or call +44 (0) 1243 843291



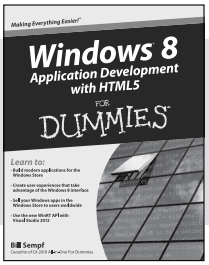
FOR DUMMIES®

Making Everything Easier!™

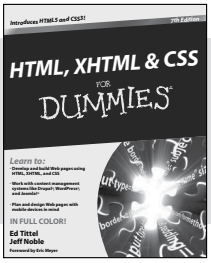
WEB DEVELOPMENT



978-1-118-38318-6

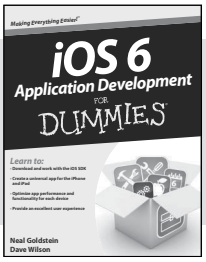


978-1-1181-7335-0

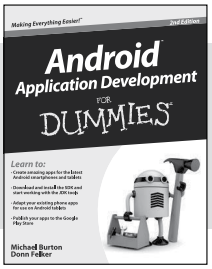


978-0-470-91659-9

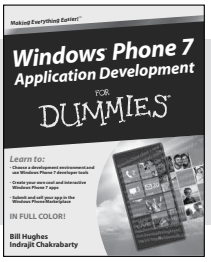
MOBILE DEVELOPMENT



978-1-1185-0880-0

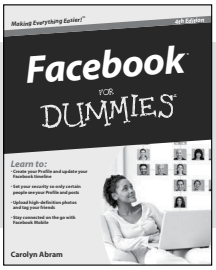


978-1-1183-8710-8

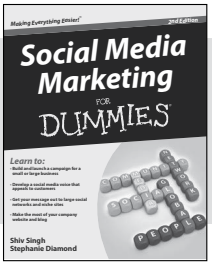


978-1-1180-2175-0

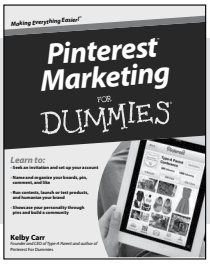
SOCIAL MEDIA



978-1-1180-9562-1



978-1-118-06514-3



978-1-118-38315-5

Android Application Development For Dummies, 2nd Edition
978-1-118-38710-8

Android Game Programming For Dummies
978-1-118-02774-5

Android Tablet Application Development For Dummies
978-1-118-09623-9

Creating Web Pages All-in-One For Dummies
978-0-470-64032-6

Facebook All-in-One For Dummies
978-1-118-17108-0

Facebook Marketing All-in-One For Dummies, 2nd Edition
978-1-118-46678-0

HTML5 For Dummies eLearning Course Access Code Card
978-1-118-45737-5

iPhone Application Development For Dummies
978-1-118-09134-0

Online Reputation Management For Dummies
978-1-118-33859-9

QuickBooks 2013 All-in-One For Dummies
978-1-118-35639-5

Scrivener For Dummies
978-1-118-31247-6

Search Engine Optimization For Dummies, 5th Edition
978-1-118-33685-4

SharePoint 2010 For Dummies, 2nd Edition
978-1-118-27381-4

Social Media Marketing eLearning Kit For Dummies
978-1-118-03470-5

Social Media Marketing For Dummies, 2nd Edition
978-1-118-06514-3

Social Media Metrics For Dummies
978-1-118-02775-2

Twitter For Dummies, 2nd Edition
978-0-470-76879-2

Web Marketing All-in-One For Dummies, 2nd Edition
978-1-118-24377-0

Available wherever books are sold. For more information or to order direct go to www.wiley.com or call +44 (0) 1243 843291

Mobile Apps FOR DUMMIES®

There's a Dummies App for This and That

With more than 200 million books in print and over 1,600 unique titles, Dummies is a global leader in how-to information. Now you can get the same great Dummies information in an App. With topics such as Wine, Spanish, Digital Photography, Certification, and more, you'll have instant access to the topics you need to know in a format you can trust.

To get information on all our Dummies apps, visit the following:

www.Dummies.com/go/mobile from your computer.

www.Dummies.com/go/iphone/apps from your phone.