HZ BOOKS
PEARSON Addison Wesley

设计模式开山之作　经典图书双语重读

设计模式

可复用面向对象软件的基础

双语版

Design Patterns: Elements of Reusable Object-Oriented Software

(美) Erich Gamma Richard Helm Ralph Johnson John Vlissides 著

李英军 马晓星 蔡敏 刘建中 译

机械工业出版社
China Machine Press

设计模式

可复用面向对象软件的基础

双语版

Design Patterns:
Elements
of Reusable Object-Oriented
Software

(美) Erich Gamma Richard Helm Ralph Johnson John Vlissides 著

李英军 马晓星 蔡敏 刘建中 译

机械工业出版社
China Machine Press

本书是软件设计领域中的经典著作，对软件技术的发展起了重要作用。本书结合设计实例从面向对象的设计中精选出23个设计模式，总结了面向对象设计中最有价值的经验，并且用简洁可复用的形式表达出来。本书分类描述了一组设计良好、表达清楚的软件设计模式，这些模式在实用环境下特别有用。本书适合大学计算机专业的学生、研究生及相关人员参考。

本书版权登记号：图字：01-2007-1743

图书在版编目（CIP）数据

设计模式：可复用面向对象软件的基础：双语版 /（美）伽玛（Gamma, E.）等著；李英军等译. -北京：机械工业出版社，2007.3
书名原文：Design Patterns: Elements of Reusable Object-Oriented Software
ISBN 978-7-111-21126-6

I. 设… II. ① 伽… ② 李… III. 面向对象语言-程序设计-英、汉 IV. TP312

中国版本图书馆CIP数据核字（2007）第033511号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：陈冀康
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2007年3月第1版第1次印刷
186mm × 240mm · 41印张
定价：69.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线（010）68326294

Preface

This book isn't an introduction to object-oriented technology or design. Many books already do a good job of that. This book assumes you are reasonably proficient in at least one object-oriented programming language, and you should have some experience in object-oriented design as well. You definitely shouldn't have to rush to the nearest dictionary the moment we mention "types" and "polymorphism," or "interface" as opposed to "implementation" inheritance.

On the other hand, this isn't an advanced technical treatise either. It's a book of **design patterns** that describes simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form.

The design patterns require neither unusual language features nor amazing programming tricks with which to astound your friends and managers. All can be implemented in standard object-oriented languages, though they might take a little more work than *ad hoc* solutions. But the extra effort invariably pays dividends in increased flexibility and reusability.

Once you understand the design patterns and have had an "Aha!" (and not just a "Huh?") experience with them, you won't ever think about object-oriented design in the same way. You'll have insights that can make your own designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place, right?

A word of warning and encouragement: Don't worry if you don't understand this book completely on the first reading. We didn't understand it all on the first writing! Remember that this isn't a book to read once and put on a shelf. We hope you'll find yourself referring to it again and again for design insights and for inspiration.

This book has had a long gestation. It has seen four countries, three of its authors' marriages, and the birth of two (unrelated) offspring. Many people have had a part in its development. Special thanks are due Bruce Anderson, Kent Beck, and André Weinand for their inspiration and advice. We also thank those who reviewed drafts

of the manuscript: Roger Bielefeld, Grady Booch, Tom Cargill, Marshall Cline, Ralph Hyre, Brian Kernighan, Thomas Laliberty, Mark Lorenz, Arthur Riel, Doug Schmidt, Clovis Tondo, Steve Vinoski, and Rebecca Wirfs-Brock. We are also grateful to the team at Addison-Wesley for their help and patience: Kate Habib, Tiffany Moore, Lisa Raffaele, Pradeepa Siva, and John Wait. Special thanks to Carl Kessler, Danny Sabbah, and Mark Wegman at IBM Research for their unflagging support of this work.

Last but certainly not least, we thank everyone on the Internet and points beyond who commented on versions of the patterns, offered encouraging words, and told us that what we were doing was worthwhile. These people include but are not limited to Jon Avotins, Steve Berczuk, Julian Berdych, Matthias Bohlen, John Brant, Allan Clarke, Paul Chisholm, Jens Coldewey, Dave Collins, Jim Coplien, Don Dwiggins, Gabriele Elia, Doug Felt, Brian Foote, Denis Fortin, Ward Harold, Hermann Hueni, Nayeem Islam, Bikramjit Kalra, Paul Keefer, Thomas Kofler, Doug Lea, Dan LaLiberte, James Long, Ann Louise Luu, Pundi Madhavan, Brian Marick, Robert Martin, Dave McComb, Carl McConnell, Christine Mingins, Hanspeter Mössenböck, Eric Newton, Marianne Ozkan, Roxsan Payette, Larry Podmolik, George Radin, Sita Ramakrishnan, Russ Ramirez, Alexander Ran, Dirk Riehle, Bryan Rosenburg, Aamod Sane, Duri Schmidt, Robert Seidl, Xin Shu, and Bill Walker.

We don't consider this collection of design patterns complete and static; it's more a recording of our current thoughts on design. We welcome comments on it, whether criticisms of our examples, references and known uses we've missed, or design patterns we should have included. You can write us care of Addison-Wesley, or send electronic mail to `design-patterns@cs.uiuc.edu`. You can also obtain softcopy for the code in the Sample Code sections by sending the message "send design pattern source" to `design-patterns-source@cs.uiuc.edu`. And now there's a Web page at `http://st-www.cs.uiuc.edu/users/patterns/DPBook/DPBook.html` for late-breaking information and updates.

Mountain View, California E.G.

Montreal, Quebec R.H.

Urbana, Illinois R.J.

Hawthorne, New York J.V.

August 1994

Foreword

All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored.

The importance of patterns in crafting complex systems has been long recognized in other disciplines. In particular, Christopher Alexander and his colleagues were perhaps the first to propose the idea of using a pattern language to architect buildings and cities. His ideas and the contributions of others have now taken root in the object-oriented software community. In short, the concept of the design pattern in software provides a key to helping developers leverage the expertise of other skilled architects.

In this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides introduce the principles of design patterns and then offer a catalog of such patterns. Thus, this book makes two important contributions. First, it shows the role that patterns can play in architecting complex systems. Second, it provides a very pragmatic reference to a set of well-engineered patterns that the practicing developer can apply to crafting his or her own specific applications.

I'm honored to have had the opportunity to work directly with some of the authors of this book in architectural design efforts. I have learned much from them, and I suspect that in reading this book, you will also.

Grady Booch

Chief Scientist, Rational Software Corporation

Guide to Readers

This book has two main parts. The first part (Chapters 1 and 2) describes what design patterns are and how they help you design object-oriented software. It includes a design case study that demonstrates how design patterns apply in practice. The second part of the book (Chapters 3, 4, and 5) is a catalog of the actual design patterns.

The catalog makes up the majority of the book. Its chapters divide the design patterns into three types: creational, structural, and behavioral. You can use the catalog in several ways. You can read the catalog from start to finish, or you can just browse from pattern to pattern. Another approach is to study one of the chapters. That will help you see how closely related patterns distinguish themselves.

You can use the references between the patterns as a logical route through the catalog. This approach will give you insight into how patterns relate to each other, how they can be combined with other patterns, and which patterns work well together. Figure 1.1 (page 12) depicts these references graphically.

Yet another way to read the catalog is to use a more problem-directed approach. Skip to Section 1.6 (page 24) to read about some common problems in designing reusable object-oriented software; then read the patterns that address these problems. Some people read the catalog through first and *then* use a problem-directed approach to apply the patterns to their projects.

If you aren't an experienced object-oriented designer, then start with the simplest and most common patterns:

- Abstract Factory (page 87)
- Adapter (139)
- Composite (163)
- Decorator (175)
- Factory Method (107)
- Observer (293)
- Strategy (315)
- Template Method (325)

It's hard to find an object-oriented system that doesn't use at least a couple of these patterns, and large systems use nearly all of them. This subset will help you understand design patterns in particular and good object-oriented design in general.

Contents

目 录

Chapter 1

Introduction

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

An analogy will help illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects"

and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically.

We all know the value of design experience. How many times have you had design *déjà-vu*—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use.

The purpose of this book is to record experience in designing object-oriented software as **design patterns**. Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

None of the design patterns in this book describes new or unproven designs. We have included only designs that have been applied more than once in different systems. Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems—neither of which is easy for novice designers to learn from. So although these designs aren't new, we capture them in a new and accessible way: as a catalog of design patterns having a consistent format.

Despite the book's size, the design patterns in it capture only a fraction of what an expert might know. It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming. It doesn't have any application domain-specific patterns. It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too.

1.1 What Is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77, page *x*]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls

and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

   The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. *Design patterns* are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations,

and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation.

Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self). We chose Smalltalk and C++ for pragmatic reasons: Our day-to-day experience has been in these languages, and they are increasingly popular.

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 331). In fact, there are enough differences between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other. (See Iterator (257) for an example.)

1.2 Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern."

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The following diagram shows a model and three views. (We've left out the controllers for simplicity.) The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.