PEARSON
Addison Wesley

Design Patterns in C#

C#设计模式

（影印版）

[美] Steven John Metsker 著

- 完全覆盖 23 个经典设计模式
- 提供使用 C# 编写的完全代码实例
- 程序设计习题有助于读者快速掌握相关技巧
- 广泛使用 UML，提高读者的 UML 使用技能

PEARSON Education

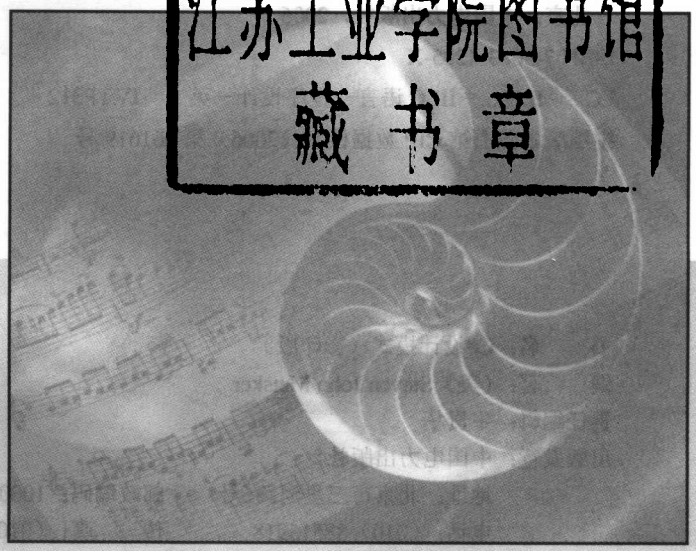
中国电力出版社
www.infopower.com.cn

Design Patterns in C#

C#设计模式

（影印版）

[美] Steven John Metsker 著



中国电力出版社
www.infopower.com.cn

Design Patterns in C# (ISBN 0-321-12697-1)
Steven John Metsker

北京市版权局著作合同登记号 图字：01-2006-3391 号

图书在版编目（CIP）数据

C#设计模式＝Design Patterns in C# /（美）麦斯科（Metsker,S.J.）著．—影印本．
—北京：中国电力出版社，2006
ISBN 7-5083-4295-X
I.C... II.麦... III.C 语言－程序设计－英文 IV.TP312
中国版本图书馆 CIP 数据核字（2006）第 061619 号

书　　名：C#设计模式（影印版）
编　　著：（美）Steven John Metsker
责任编辑：牛贵华
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开本尺寸：185×233　　印　　张：29.75
书　　号：ISBN 7-5083-4295-X
版　　次：2006 年 7 月北京第 1 版　　2006 年 7 月第 1 次印刷
定　　价：49.00 元

To Emma-Kate and Sarah-Jane,
Two cherubim aswirl
To Alison, the nurturing heart
To heartstring-strumming girls.

Interface Patterns

ADAPTER (page 19) Provide the interface that a client expects, using the services of a class with a different interface.

FACADE (page 35) Provide an interface that makes a subsystem easy to use.

COMPOSITE (page 49) Allow clients to treat individual objects and compositions of objects uniformly.

BRIDGE (page 65) Decouple an abstraction (a class that relies on abstract operations) from the implementation of its abstract operations so that the abstraction and its implementation can vary independently.

Responsibility Patterns

SINGLETON (page 83) Ensure that a class has only one instance, and provide a global point of access to it.

OBSERVER (page 89) Define a one-to-many dependency among objects so that when one object changes state, all of its dependents are notified and updated automatically.

MEDIATOR (page 109) Define an object that encapsulates the way a set of objects interact. This keeps the objects from referring to each other explicitly and lets you vary their interaction independently.

PROXY (page 123) Provide a placeholder for another object to control access to it.

CHAIN OF RESPONSIBILITY (page 139) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

FLYWEIGHT (page 147) Use sharing to support large numbers of fine-grained objects efficiently.

Construction Patterns

BUILDER (page 163) Move the construction logic for an object outside the class to instantiate.

FACTORY METHOD (page 171) Define the interface for creating an object while retaining control of which class to instantiate.

ABSTRACT FACTORY (page 179) Provide for the creation of a family of related or dependent objects.

PROTOTYPE (page 191) Provide new objects by copying an example.

MEMENTO (page 197) Provide for the storage and restoration of an object's state.

Operation Patterns

Extension Patterns

Back Matter

PREFACE

It seems a long time ago (two years!) that I received the initial encouragement for this book from Paul Becker, an editor at the time with Addison-Wesley. I remain grateful to Paul for his help, and to his successor, John Neidhart, who took over as editor when Paul left. I am also grateful for the encouragement of John Vlissides, who is the Patterns Series editor and who has been a supporter of mine for all three of my books.

John Vlissides is also, of course, one of the four authors of *Design Patterns*. John and his co-authors—Erich Gamma, Ralph Johnson, and Richard Helm—produced the work that not only established a list of important patterns that every developer should know, but also set a standard for quality and clarity that I have aspired to attain in my own writing.

In addition to relying heavily on *Design Patterns*, I have benefited from the use of many other books; see Bibliography on 439. In particular, I have depended on *The Unified Modeling Language User Guide* [Booch] for its clear explanations of UML. For concise and accurate help on C# topics, I have consulted *C# Essentials* [Albahari] almost daily. I have also repeatedly drawn on the insights of *C# and the .NET Platform* [Troelsen], and for realistic fireworks examples, I have consistently consulted *The Chemistry of Fireworks* [Russell].

As the present book began to take shape, several excellent reviewers helped to guide its progress. I would like to thank Bill Wake for his early reviews. Bill never ceases to amaze me in his ability to catch the subtlest errors while simultaneously providing advice on overall direction, content, and style. I would also like to thank Steve Berczuk and Neil Harrison. In particular, they hit on the same key point that the book needed more introductory material for each pattern. Their comments drove me to rework the entire book. It is much stronger now because of their advice.

With the help of editors and reviewers, I was able to write this book; but, the text of a book is just the beginning. I would like to thank Nick Radhuber and the entire production staff for their hard work and dedication. Their work renders text into what is to this day its most usable form—a book!

Steve Metsker (Steve.Metsker@acm.org)

Contents

Part 3: Construction Patterns

CHAPTER 1

INTRODUCTION

This book is for developers who know C# and want to improve their skills as designers. This book covers the same list of techniques as the classic book *Design Patterns*, written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book (*Design Patterns in C#*) provides examples in C# and in a .NET setting. This book also includes many "challenges," exercises designed to help strengthen your ability to apply design patterns in your own C# programs.

Why Patterns?

A *pattern* is a way of doing something, a way of pursuing an intent, a technique. The idea of capturing effective techniques applies to many endeavors: making food, making fireworks, making software, and to other crafts. In any new craft that is starting to mature, the people working on it will begin to find common, effective methods for achieving aims and solving problems in various contexts. The community of people that practice a craft usually invent jargon that helps them talk about their craft. Some of this jargon will refer to patterns, established techniques for achieving certain aims. As a craft and its jargon grows, writers begin to play an important role. Writers document a craft's patterns, helping to standardize the jargon and publicize effective techniques.

Christopher Alexander was one of the first writers to encapsulate a craft's best practices by documenting its patterns. His work relates to architecture—of buildings, not software. In *A Pattern Language: Towns, Buildings, Construction*, Alexander provides patterns for architecting successful buildings and towns. His writing is powerful and has influenced the software community, partially because of the way he looks at intent.

You might state the intent of architectural patterns as something like "to design buildings." But Alexander makes it clear that the intent of architectural patterns is to serve and inspire the people who will occupy buildings and towns. Alexander's work showed that patterns are an excellent

way to capture and convey the wisdom of a craft. He also established that properly perceiving and documenting the intent of a craft is a critical, philosophical, and elusive challenge.

The software community has resonated with the patterns approach and has created many books that document patterns of software development. These books record best practices for software process, software analysis, high-level architecture, and class-level design, and new patterns books appear every year. If you are choosing a book to read about patterns, you should spend some time reading reviews of available books and try to select the book that will help you the most.

Why Design Patterns?

A *design pattern* is a pattern—a way to pursue an intent—that uses classes and their methods in an object-oriented (OO) language. Developers often start thinking about design after learning a programming language and writing code for a while. You might notice that someone else's code seems simpler and works better than yours does and you might wonder how that developer achieves such simplicity. Design patterns are a level up from code and typically show how to achieve a goal using a few classes. Other people have discovered how to program effectively in OO languages. If you want to become a powerful C# programmer, then you should study design patterns, especially those in this book—the same patterns that *Design Patterns* explains.

Design Patterns describes 23 design patterns. Many other books on design patterns have followed, so that there are at least 100 design patterns worth knowing. The 23 design patterns that Gamma, Helm, Johnson, and Vlissides placed in *Design Patterns* are probably not absolutely the most useful 23 design patterns to know. On the other hand, these patterns are probably among the 100 most useful patterns. Unfortunately, there is no set of criteria that establishes the value of a pattern, and so the identity of the other 77 patterns in the top 100 is not yet established. Fortunately, the authors of *Design Patterns* chose well, and the patterns they documented are certainly worth learning. Learning these patterns will also serve as a foundation as you branch out and begin learning patterns from other sources.

GoF

You may have noted the potential confusion between "design patterns" the topic and *Design Patterns* the book. The topic and the book title *sound* alike, so to distinguish them, many speakers and some writers refer to the book as the "Gang of Four" book, or the "GoF" book, referring to the number of its authors. In print, it is not so confusing that *Design Patterns* refers to the book and "design patterns" refers to the topic. Accordingly, this book avoids using the term "GoF."

Why C#?

This book gives its examples in C#, the OO language at the center of Microsoft's .NET framework. The .NET framework is a suite of products for developing and managing systems with tiered, OO architectures.

The .NET framework is important simply because of Microsoft's size. Also, .NET aligns at a high level with industry thinking about how to compose an architecture. You can compare most .NET components with competing products from vendors that implement software according to the Java™ 2 Platform, Enterprise Edition (J2EE) specification. But note that J2EE is a specification, not a product, while .NET is the product lineup of one company and its partners.

At a superficial level, C# is important because it is the central OO language for developing in .NET. C# is also important because, like Java, it is a *consolidation language*, designed to absorb the strengths of earlier languages. This consolidation has fueled the popularity of Java and C# and helps ensure that future languages will evolve from these languages rather than depart radically from them. Your investment in C# will almost surely yield value in any language that supplants C#.

The patterns in *Design Patterns* apply to C# because, like Smalltalk, C++, and Java, C# follows a class/instance paradigm. C# is much more similar to Smalltalk and C++ than it is to, say, Prolog or Self. Although competing paradigms are important, the class/instance paradigm appears to be the most practical next step in applied computing. This book uses C# because of the popularity of C# and .NET, and because C# appears to lie along the evolutionary path of languages that we will use for decades ahead.

UML

Where this book's challenges (exercises) have solutions in code, this book uses C#. Many of the challenges ask you to draw a diagram of how classes, packages, and other elements relate. You can use any notation