原版风暴·软件工程系列

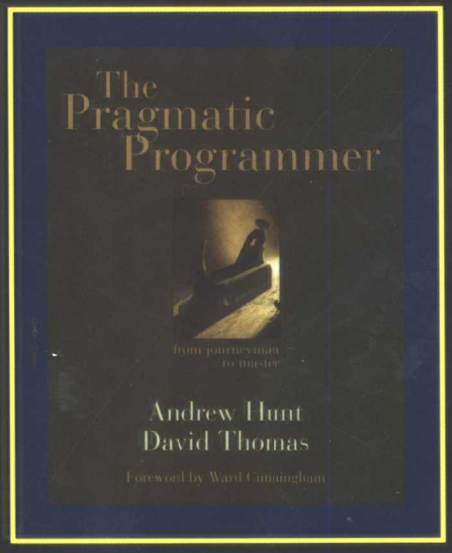
The Pragmatic Programmer

From Journeyman to Master

程序员修炼之道

（影印版）

[美] Andrew Hunt　David Thomas　著



- 致力成为高效多产程序员的绝对选择
- 大师 Kent Beck、Martin Fowler 大力推荐
- 软件开发和管理人员必读经典

中国电力出版社
www.infopower.com.cn

The Pragmatic Programmer

程序员修炼之道

原版风暴 • 软件工程系列

The Pragmatic Programmer

From Journeyman to Master

程序员修炼之道

（影印版）

[美] Andrew Hunt　David Thomas　著

中国电力出版社

The Pragmatic Programmer(ISBN 0-201-61622-X)
Andrew Hunt,David Thomas

北京市版权局著作合同登记号：图字：01-2003-2444

图书在版编目（CIP）数据

程序员修炼之道 /（美）亨特，（美）索马斯著．影印本．—北京：中国电力出版社，2003
（原版风暴・软件工程系列）
ISBN 7-5083-0798-4

Ⅰ.程... Ⅱ.①亨... ②索... Ⅲ.程序设计—方法—英文 Ⅳ.TP311.11
出版者：中国电力出版社（北京西城区三里河路 6 号，邮编 100044）

责任编辑：陈维宁
丛 书 名： 原版风暴・软件工程系列
书　　名： 程序员修炼之道（影印版）
编　　著：（美）Andrew Hunt&David Thomas
出 版 者：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： 北京地矿印刷厂
发 行 者： 新华书店总店北京发行所
开　　本： 787×1092 1/16　**印　　张：** 22
书　　号： ISBN 7-5083-0798-4
版　　次： 2003年8月北京第一版
印　　次： 2003年8月第一次印刷
定　　价： 39.00 元

What others in the trenches say about *The Pragmatic Programmer*...

"The cool thing about this book is that it's great for keeping the programming process fresh. *[The book]* helps you to continue to grow and clearly comes from people who have been there."

► **Kent Beck**, author of *Extreme Programming Explained: Embrace Change*

"I found this book to be a great mix of solid advice and wonderful analogies!"

► **Martin Fowler**, author of *Refactoring* and *UML Distilled*

"I would buy a copy, read it twice, then tell all my colleagues to run out and grab a copy. This is a book I would never loan because I would worry about it being lost."

► **Kevin Ruland**, Management Science, MSG-Logistics

"The wisdom and practical experience of the authors is obvious. The topics presented are relevant and useful. . . . By far its greatest strength for me has been the outstanding analogies—tracer bullets, broken windows, and the fabulous helicopter-based explanation of the need for orthogonality, especially in a crisis situation. I have little doubt that this book will eventually become an excellent source of useful information for journeymen programmers and expert mentors alike."

► **John Lakos**, author of *Large-Scale C++ Software Design*

"This is the sort of book I will buy a dozen copies of when it comes out so I can give it to my clients."

► **Eric Vought,** Software Engineer

"Most modern books on software development fail to cover the basics of what makes a great software developer, instead spending their time on syntax or technology where in reality the greatest leverage possible for any software team is in having talented developers who really know their craft well. An excellent book."

► **Pete McBreen,** Independent Consultant

"Since reading this book, I have implemented many of the practical suggestions and tips it contains. Across the board, they have saved my company time and money while helping me get my job done quicker! This should be a desktop reference for everyone who works with code for a living."

► **Jared Richardson,** Senior Software Developer,
iRenaissance, Inc.

"I would like to see this issued to every new employee at my company. . . ."

► **Chris Cleeland,** Senior Software Engineer,
Object Computing, Inc.

The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt
David Thomas

Foreword

As a reviewer I got an early opportunity to read the book you are holding. It was great, even in draft form. Dave Thomas and Andy Hunt have something to say, and they know how to say it. I saw what they were doing and I knew it would work. I asked to write this foreword so that I could explain why.

Simply put, this book tells you how to program in a way that you can follow. You wouldn't think that that would be a hard thing to do, but it is. Why? For one thing, not all programming books are written by programmers. Many are compiled by language designers, or the journalists who work with them to promote their creations. Those books tell you how to *talk* in a programming language—which is certainly important, but that is only a small part of what a programmer does.

What does a programmer do besides talk in programming language? Well, that is a deeper issue. Most programmers would have trouble explaining what they do. Programming is a job filled with details, and keeping track of those details requires focus. Hours drift by and the code appears. You look up and there are all of those statements. If you don't think carefully, you might think that programming is just typing statements in a programming language. You would be wrong, of course, but you wouldn't be able to tell by looking around the programming section of the bookstore.

In *The Pragmatic Programmer* Dave and Andy tell us how to program in a way that we can follow. How did they get so smart? Aren't they just as focused on details as other programmers? The answer is that they paid attention to what they were doing while they were doing it—and then they tried to do it better.

Imagine that you are sitting in a meeting. Maybe you are thinking that the meeting could go on forever and that you would rather be programming. Dave and Andy would be thinking about why they were

having the meeting, and wondering if there is something else they could do that would take the place of the meeting, and deciding if that something could be automated so that the work of the meeting just happens in the future. Then they would do it.

That is just the way Dave and Andy think. That meeting wasn't something keeping them from programming. It *was* programming. And it was programming that could be improved. I know they think this way because it is tip number two: Think About Your Work.

So imagine that these guys are thinking this way for a few years. Pretty soon they would have a collection of solutions. Now imagine them using their solutions in their work for a few more years, and discarding the ones that are too hard or don't always produce results. Well, that approach just about defines *pragmatic*. Now imagine them taking a year or two more to write their solutions down. You might think, *That information would be a gold mine*. And you would be right.

The authors tell us how they program. And they tell us in a way that we can follow. But there is more to this second statement than you might think. Let me explain.

The authors have been careful to avoid proposing a theory of software development. This is fortunate, because if they had they would be obliged to warp each chapter to defend their theory. Such warping is the tradition in, say, the physical sciences, where theories eventually become laws or are quietly discarded. Programming on the other hand has few (if any) laws. So programming advice shaped around wanna-be laws may sound good in writing, but it fails to satisfy in practice. This is what goes wrong with so many methodology books.

I've studied this problem for a dozen years and found the most promise in a device called a *pattern language*. In short, a *pattern* is a solution, and a pattern language is a system of solutions that reinforce each other. A whole community has formed around the search for these systems.

This book is more than a collection of tips. It is a pattern language in sheep's clothing. I say that because each tip is drawn from experience, told as concrete advice, and related to others to form a system. These are the characteristics that allow us to learn and follow a pattern language. They work the same way here.

You can follow the advice in this book because it is concrete. You won't find vague abstractions. Dave and Andy write directly for you, as if each tip was a vital strategy for energizing your programming career. They make it simple, they tell a story, they use a light touch, and then they follow that up with answers to questions that will come up when you try.

And there is more. After you read ten or fifteen tips you will begin to see an extra dimension to the work. We sometimes call it *QWAN*, short for the *quality without a name*. The book has a philosophy that will ooze into your consciousness and mix with your own. It doesn't preach. It just tells what works. But in the telling more comes through. That's the beauty of the book: It embodies its philosophy, and it does so unpretentiously.

So here it is: an easy to read—and use—book about the whole practice of programming. I've gone on and on about why it works. You probably only care that it does work. It does. You will see.

—Ward Cunningham

Preface

This book will help you become a better programmer.

It doesn't matter whether you are a lone developer, a member of a large project team, or a consultant working with many clients at once. This book will help you, as an individual, to do better work. This book isn't theoretical—we concentrate on practical topics, on using your experience to make more informed decisions. The word *pragmatic* comes from the Latin *pragmaticus*—"skilled in business"—which itself is derived from the Greek $\pi\rho\alpha\tau\tau\epsilon\iota\nu$, meaning "to do." This is a book about doing.

Programming is a craft. At its simplest, it comes down to getting a computer to do what you want it to do (or what your user wants it to do). As a programmer, you are part listener, part advisor, part interpreter, and part dictator. You try to capture elusive requirements and find a way of expressing them so that a mere machine can do them justice. You try to document your work so that others can understand it, and you try to engineer your work so that others can build on it. What's more, you try to do all this against the relentless ticking of the project clock. You work small miracles every day.

It's a difficult job.

There are many people offering you help. Tool vendors tout the miracles their products perform. Methodology gurus promise that their techniques guarantee results. Everyone claims that their programming language is the best, and every operating system is the answer to all conceivable ills.

Of course, none of this is true. There are no easy answers. There is no such thing as a *best* solution, be it a tool, a language, or an operating system. There can only be systems that are more appropriate in a particular set of circumstances.

This is where pragmatism comes in. You shouldn't be wedded to any particular technology, but have a broad enough background and experience base to allow you to choose good solutions in particular situations. Your background stems from an understanding of the basic principles of computer science, and your experience comes from a wide range of practical projects. Theory and practice combine to make you strong.

You adjust your approach to suit the current circumstances and environment. You judge the relative importance of all the factors affecting a project and use your experience to produce appropriate solutions. And you do this continuously as the work progresses. Pragmatic Programmers get the job done, and do it well.

Who Should Read This Book?

This book is aimed at people who want to become more effective and more productive programmers. Perhaps you feel frustrated that you don't seem to be achieving your potential. Perhaps you look at colleagues who seem to be using tools to make themselves more productive than you. Maybe your current job uses older technologies, and you want to know how newer ideas can be applied to what you do.

We don't pretend to have all (or even most) of the answers, nor are all of our ideas applicable in all situations. All we can say is that if you follow our approach, you'll gain experience rapidly, your productivity will increase, and you'll have a better understanding of the entire development process. And you'll write better software.

What Makes a Pragmatic Programmer?

Each developer is unique, with individual strengths and weaknesses, preferences and dislikes. Over time, each will craft his or her own personal environment. That environment will reflect the programmer's individuality just as forcefully as his or her hobbies, clothing, or haircut. However, if you're a Pragmatic Programmer, you'll share many of the following characteristics:

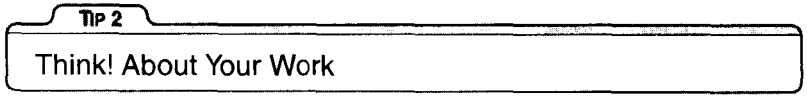
- **Early adopter/fast adapter.** You have an instinct for technologies and techniques, and you love trying things out. When given some-

thing new, you can grasp it quickly and integrate it with the rest of your knowledge. Your confidence is born of experience.

- **Inquisitive.** You tend to ask questions. *That's neat—how did you do that? Did you have problems with that library? What's this BeOS I've heard about? How are symbolic links implemented?* You are a pack rat for little facts, each of which may affect some decision years from now.
- **Critical thinker.** You rarely take things as given without first getting the facts. When colleagues say "because that's the way it's done," or a vendor promises the solution to all your problems, you smell a challenge.
- **Realistic.** You try to understand the underlying nature of each problem you face. This realism gives you a good feel for how difficult things are, and how long things will take. Understanding for yourself that a process *should* be difficult or *will* take a while to complete gives you the stamina to keep at it.
- **Jack of all trades.** You try hard to be familiar with a broad range of technologies and environments, and you work to keep abreast of new developments. Although your current job may require you to be a specialist, you will always be able to move on to new areas and new challenges.

We've left the most basic characteristics until last. All Pragmatic Programmers share them. They're basic enough to state as tips:

We feel that there is no point in developing software unless you care about doing it well.



In order to be a Pragmatic Programmer, we're challenging you to think about what you're doing while you're doing it. This isn't a one-time audit of current practices—it's an ongoing critical appraisal of every

decision you make, every day, and on every development. Never run on auto-pilot. Constantly be thinking, critiquing your work in real time. The old IBM corporate motto, THINK!, is the Pragmatic Programmer's mantra.

If this sounds like hard work to you, then you're exhibiting the *realistic* characteristic. This is going to take up some of your valuable time—time that is probably already under tremendous pressure. The reward is a more active involvement with a job you love, a feeling of mastery over an increasing range of subjects, and pleasure in a feeling of continuous improvement. Over the long term, your time investment will be repaid as you and your team become more efficient, write code that's easier to maintain, and spend less time in meetings.

Individual Pragmatists, Large Teams

Some people feel that there is no room for individuality on large teams or complex projects. "Software construction is an engineering discipline," they say, "that breaks down if individual team members make decisions for themselves."

We disagree.

The construction of software *should* be an engineering discipline. However, this doesn't preclude individual craftsmanship. Think about the large cathedrals built in Europe during the Middle Ages. Each took thousands of person-years of effort, spread over many decades. Lessons learned were passed down to the next set of builders, who advanced the state of structural engineering with their accomplishments. But the carpenters, stonecutters, carvers, and glass workers were all craftspeople, interpreting the engineering requirements to produce a whole that transcended the purely mechanical side of the construction. It was their belief in their individual contributions that sustained the projects:

> *We who cut mere stones must always be envisioning cathedrals.*
> **— Quarry worker's creed**

Within the overall structure of a project there is always room for individuality and craftsmanship. This is particularly true given the current state of software engineering. One hundred years from now, our engineering may seem as archaic as the techniques used by medieval

cathedral builders seem to today's civil engineers, while our craftsmanship will still be honored.

It's a Continuous Process

> *A tourist visiting England's Eton College asked the gardener how he got the lawns so perfect. "That's easy," he replied, "You just brush off the dew every morning, mow them every other day, and roll them once a week."*
>
> *"Is that all?" asked the tourist.*
>
> *"Absolutely," replied the gardener. "Do that for 500 years and you'll have a nice lawn, too."*

Great lawns need small amounts of daily care, and so do great programmers. Management consultants like to drop the word *kaizen* in conversations. "Kaizen" is a Japanese term that captures the concept of continuously making many small improvements. It was considered to be one of the main reasons for the dramatic gains in productivity and quality in Japanese manufacturing and was widely copied throughout the world. Kaizen applies to individuals, too. Every day, work to refine the skills you have and to add new tools to your repertoire. Unlike the Eton lawns, you'll start seeing results in a matter of days. Over the years, you'll be amazed at how your experience has blossomed and your skills have grown.

How the Book Is Organized

This book is written as a collection of short sections. Each section is self-contained, and addresses a particular topic. You'll find numerous cross references, which help put each topic in context. Feel free to read the sections in any order—this isn't a book you need to read front-to-back.

Occasionally you'll come across a box labeled *Tip nn* (such as Tip 1, "Care About Your Craft" on page xix). As well as emphasizing points in the text, we feel the tips have a life of their own—we live by them daily. You'll find a summary of all the tips on a pull-out card inside the back cover.

Appendix A contains a set of resources: the book's bibliography, a list of URLs to Web resources, and a list of recommended periodicals, books, and professional organizations. Throughout the book you'll find references to the bibliography and to the list of URLs—such as [KP99] and [URL 18], respectively.

We've included exercises and challenges where appropriate. Exercises normally have relatively straightforward answers, while the challenges are more open-ended. To give you an idea of our thinking, we've included our answers to the exercises in Appendix B, but very few have a single *correct* solution. The challenges might form the basis of group discussions or essay work in advanced programming courses.

What's in a Name?

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."

► **Lewis Carroll, *Through the Looking-Glass***

Scattered throughout the book you'll find various bits of jargon—either perfectly good English words that have been corrupted to mean something technical, or horrendous made-up words that have been assigned meanings by computer scientists with a grudge against the language. The first time we use each of these jargon words, we try to define it, or at least give a hint to its meaning. However, we're sure that some have fallen through the cracks, and others, such as *object* and *relational database*, are in common enough usage that adding a definition would be boring. If you *do* come across a term you haven't seen before, please don't just skip over it. Take time to look it up, perhaps on the Web, or maybe in a computer science textbook. And, if you get a chance, drop us an e-mail and complain, so we can add a definition to the next edition.

Having said all this, we decided to get revenge against the computer scientists. Sometimes, there are perfectly good jargon words for concepts, words that we've decided to ignore. Why? Because the existing jargon is normally restricted to a particular problem domain, or to a particular phase of development. However, one of the basic philosophies of this book is that most of the techniques we're recommending are universal: modularity applies to code, designs, documentation, and team

organization, for instance. When we wanted to use the conventional jargon word in a broader context, it got confusing—we couldn't seem to overcome the baggage the original term brought with it. When this happened, we contributed to the decline of the language by inventing our own terms.

Source Code and Other Resources

Most of the code shown in this book is extracted from compilable source files, available for download from our Web site:

```
www.pragmaticprogrammer.com
```

There you'll also find links to resources we find useful, along with updates to the book and news of other Pragmatic Programmer developments.

Send Us Feedback

We'd appreciate hearing from you. Comments, suggestions, errors in the text, and problems in the examples are all welcome. E-mail us at

```
ppbook@pragmaticprogrammer.com
```

Acknowledgments

When we started writing this book, we had no idea how much of a team effort it would end up being.

Addison-Wesley has been brilliant, taking a couple of wet-behind-the-ears hackers and walking us through the whole book-production process, from idea to camera-ready copy. Many thanks to John Wait and Meera Ravindiran for their initial support, Mike Hendrickson, our enthusiastic editor (and a mean cover designer!), Lorraine Ferrier and John Fuller for their help with production, and the indefatigable Julie DeBaggis for keeping us all together.

Then there were the reviewers: Greg Andress, Mark Cheers, Chris Cleeland, Alistair Cockburn, Ward Cunningham, Martin Fowler, Thanh T. Giang, Robert L. Glass, Scott Henninger, Michael Hunter, Brian