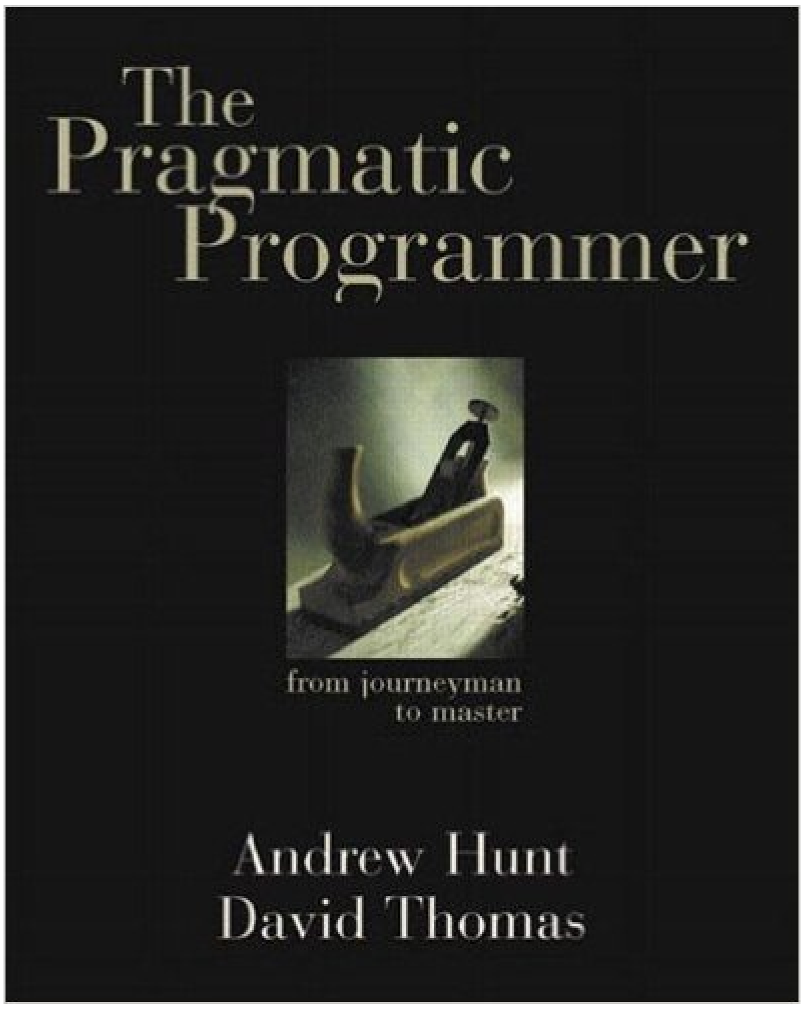
The Pragmatic Programmer: From Journeyman To Master



Synopsis

-- Ward Cunningham Straight from the programming trenches, The Pragmatic Programmer cuts through the increasing specialization and technicalities of modern software development to examine the core process--taking a requirement and producing working, maintainable code that delights its users. It covers topics ranging from personal responsibility and career development to architectural techniques for keeping your code flexible and easy to adapt and reuse. Read this book, and youll learn how to *Fight software rot; *Avoid the trap of duplicating knowledge; *Write flexible, dynamic, and adaptable code; *Avoid programming by coincidence; *Bullet-proof your code with contracts, assertions, and exceptions; *Capture real requirements; *Test ruthlessly and effectively; *Delight your users; *Build teams of pragmatic programmers; and *Make your developments more precise with automation. Written as a series of self-contained sections and filled with entertaining anecdotes, thoughtful examples, and interesting analogies, The Pragmatic Programmer illustrates the best practices and major pitfalls of many different aspects of software development. Whether youre a new coder, an experienced programm

Book Information

Paperback: 352 pages
Publisher: Addison-Wesley Professional; 1 edition (October 30, 1999)
Language: English
ISBN-10: 020161622X
ISBN-13: 978-0201616224
Product Dimensions: 7.3 x 0.9 x 9.1 inches
Shipping Weight: 1.3 pounds (View shipping rates and policies)
Average Customer Review: 4.5 out of 5 stars   347 customer reviews
Best Sellers Rank: #10,526 in Books (See Top 100 in Books)   #47 inÃ Â Books > Computers & Technology > Software   #50 inÃ Â Books > Textbooks > Computer Science > Programming Languages   #84 inÃ Â Books > Computers & Technology > Programming

Customer Reviews

Programmers are craftspeople trained to use a certain set of tools (editors, object managers, version trackers) to generate a certain kind of product (programs) that will operate in some environment (operating systems on hardware assemblies). Like any other craft, computer programming has spawned a body of wisdom, most of which isn’t taught at universities or in certification classes. Most programmers arrive at the so-called tricks of the trade over time, through independent experimentation. In The Pragmatic Programmer, Andrew Hunt and David Thomas codify many of the truths they've discovered during their respective careers as designers of software and writers of code. Some of the authors' nuggets of pragmatism are concrete, and the path to their implementation is clear. They advise readers to learn one text editor, for example, and use it for everything. They also recommend the use of version-tracking software for even the smallest projects, and promote the merits of learning regular expression syntax and a text-manipulation language. Other (perhaps more valuable) advice is more light-hearted. In the debugging section, it is noted that, "if you see hoof prints think horses, not zebras." That is, suspect everything, but start looking for problems in the most obvious places. There are recommendations for making estimates of time and expense, and for integrating testing into the development process. You'll want a copy of The Pragmatic Programmer for two reasons: it displays your own accumulated wisdom more cleanly than you ever bothered to state it, and it introduces you to methods of work that you may not yet have considered. Working programmers will enjoy this book. --David Wall Topics covered: A useful approach to software design and construction that allows for efficient, profitable development of high-quality products. Elements of the approach include specification development, customer relations, team management, design practices, development tools, and testing procedures. This approach is presented with the help of anecdotes and technical problems.

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

The good:- The advice on metaprogramming is probably the strongest part of the book. I really liked the clarity of the examples. The key idea is that you should program the structure and methods in code, and provide the detailed implementation (business rules, algorithms etc.) as data.- The advice on code generators, little languages and plain text interfaces is solid advice worth reiterating and the sections on these are good. However, I preferred the treatment offered by Jon Bentley in his really excellent Programming Pearls books- The section on automation is a healthy reminder that programmers should have a "do it once, or automate" philosophy- The section on contract-driven development is worthy and echoes current practice. Jon Bentley's version is a more theoretical, conceptual approach, less polluted by particular industry practices.- The sections on test-driven development is fine, but the ideas are much better handled by Kent Beck's booksThe not so good:- The chapters on Pragmatism, Specification, Team Work and Discipline fit nicely with the general themes of the book, but say very little of practical value. If you are interested in operating in a team environment, this isn't the book you are looking for.- The section on programming methodologies is so wishy-washy it's hard to know what advice if any to take away from it- References to specific resources are somewhat out of date (the book was written 15 years ago)The Pragmatic Programmer is worth the price, but if you are thinking of buying this book because you are a relatively new programmer and are looking for advice, I would strongly suggest first reading the much better books:Jon Bentley's Programming Pearls and More Programming PearlsKernighan and Pike's The Practice of ProgrammingKent Beck's Extreme Programming ExplainedAnother book you might be interested in is Susan Lammers, Programmers at Work. Although it is very old (1989), its interesting that most of the programmers interviewed discuss exactly the same themes picked up in The Pragmatic Programmer.In case you were also thinking about buying a copy of Code Complete, which is also often recommended to new programmers, also hold off - its far too long and a great deal of it feels like a padded-out outline.

My coworkers have a book group every other week for this book and it generates so much discussion that I don't think we'll ever finish it. Our current pace is 2-4 sections (1/2 chapter) every two weeks. I feel like a lot of this stuff was covered in Code Complete 2 which I read several years ago but there’s still new ground being covered and a refresher is definitely warranted for some of the topics.I think what’s most important to me is the interesting discussions we have while discussing this book and the introspection it brings to our current project.

Really good book; it was suggested me by a fellow coworker; and I must say that it was spot on.It won’t change your life, if you have a decent experience, but even when you think to know about a specific topic, you realize that the book give you that extra bit. Some concepts are purely logic applied to software engineering; but what the author add is a ton of extras that change that "yeah, I know that", into "we should start to implement things in this way".The difference is purely in your attitude toward your job and your habits (and potentially your team-mate); you may know something, but this book push you to not just dismiss it as "good practice", but as something to use actively in your day by day work.Plenty of examples; plenty of practical examples; either if you start now or if you are a seasoned senior; give it a read. I can assure that you will get for sure, something out of it that you didn’t know.

I liked it. The book describes the features of a "pragmatic programmer", a programmer who cares deeply for his/her profession and the quality he/she produces. If you’ve been a software developer for 5+ years, you might already be familiar with most (if not all) of the topics in this book; the importance of self-improvement, refactoring, clean code, communication, etc. Nevertheless, the book is still enjoyable and reminds you of the important aspects of programming.

Tip 15: Use Tracer Bullets to Find the Target. That’s among 70 "tips" you’ll get from Hunt and Thomas, and one of the most important. I used to think you had to plan everything ahead of time down to APIs and data structures and then go program it like an automaton. Both this book and _Extreme Programming_ by Beck explain why this is a flawed idea in the face of the plastic medium of software. You often don’t know what you want, you’ll often learn a lot by implementing the first module, or the customer may change their mind about what they want (especially after seeing early versions). I went from academia to a start-up, and this book was a very good outline of how to program in a real world setting. My boss, another former academic, also read it and loved it. You might not learn anything from this book if you’re already an old hand at professional programming on a great team. Or maybe it’ll just ring the I-knew-that-but-now-I-have-a-name-for-it bell. The discussion of rubber ducking is a good example (just nodding your head behind someone while they explain their code to you and debug it themselves). The discussion of the stone soup phenomenon is another case (read the book to see how it applies). My other favorites among the tips include: don’t live with broken windows; make quality a requirements issue; don’t repeat yourself; there are no final decisions (strongly related to tracer bullets); keep knowledge in plain text; use the power of command shells (but don’t bother reading Neal Stephenson’s book on the subject even if you love his other books as much as I do); use a single editor well; "select" isn’t broken (think horses not zebras); design with contracts; test your estimates; design to test; test your software or your users will; test early. test often. test automatically; find bugs once; and finally, sign your work. (I wish software designers could do this a little more literally; I love how hardware designers get to etch cool logos on circuit boards.)

Download to continue reading...