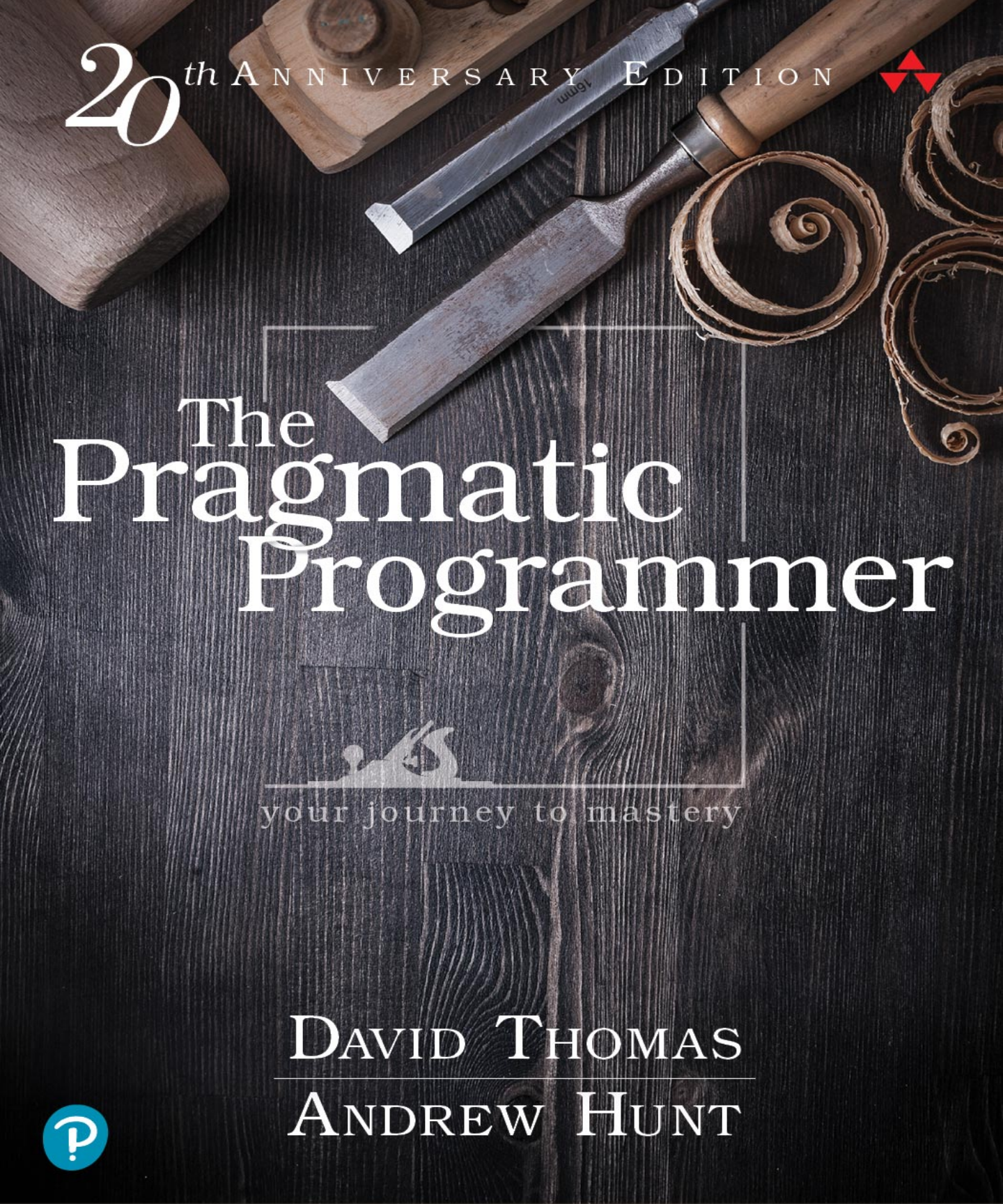
20th Anniversary Edition
The Pragmatic Programmer
your journey to mastery
David Thomas
Andrew Hunt

Praise for the second edition of *The Pragmatic Programmer*

Some say that with *The Pragmatic Programmer*, Andy and Dave captured lightning in a bottle; that it's unlikely anyone will soon write a book that can move an entire industry as it did. Sometimes, though, lightning does strike twice, and this book is proof. The updated content ensures that it will stay at the top of "best books in software development" lists for another 20 years, right where it belongs.

➤ **VM (Vicky) Brasseur**
Director of Open Source Strategy, Juniper Networks

If you want your software to be easy to modernize and maintain, keep a copy of *The Pragmatic Programmer* close. It's filled with practical advice, both technical and professional, that will serve you and your projects well for years to come.

➤ **Andrea Goulet**
CEO, Corgibytes; Founder, LegacyCode.Rocks

The Pragmatic Programmer is the one book I can point to that completely dislodged the existing trajectory of my career in software and pointed me in the direction of success. Reading it opened my mind to the possibilities of being a craftsman, not just a cog in a big machine. One of the most significant books in my life.

➤ **Obie Fernandez**
Author, *The Rails Way*

First-time readers can look forward to an enthralling induction into the modern world of software practice, a world that the first edition played a major role in shaping. Readers of the first edition will rediscover here the insights and practical wisdom that made the book so significant in the first place, expertly curated and updated, along with much that's new.

➤ **David A. Black**
Author, *The Well-Grounded Rubyist*

I have an old paper copy of the original *Pragmatic Programmer* on my bookshelf. It has been read and re-read and a long time ago it changed everything about how I approached my job as a programmer. In the new edition everything and nothing has changed: I now read it on my iPad and the code examples use modern programming languages—but the underlying concepts, ideas, and attitudes are timeless and universally applicable. Twenty years later, the book is as relevant as ever. It makes me happy to know that current and future developers will have the same opportunity to learn from Andy and Dave's profound insights as I did back in the day.

➤ **Sandy Mamoli**
Agile coach, author of *How Self-Selection Lets People Excel*

Twenty years ago, the first edition of *The Pragmatic Programmer* completely changed the trajectory of my career. This new edition could do the same for yours.

➤ **Mike Cohn**
Author of *Succeeding with Agile*, *Agile Estimating and Planning*, and *User Stories Applied*

The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Dave Thomas
Andy Hunt

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019944178

Cover images: Mihalec/Shutterstock, Stockish/Shutterstock

ISBN-13: 978-0-13-595705-9
ISBN-10: 0-13-595705-2

ScoutAutomatedPrintCode

For Juliet and Ellie,
Zachary and Elizabeth,
Henry and Stuart

Contents

Foreword

I remember when Dave and Andy first tweeted about the new edition of this book. It was big news. I watched as the coding community responded with excitement. My feed buzzed with anticipation. After twenty years, *The Pragmatic Programmer* is just as relevant today as it was back then.

It says a lot that a book with such history had such a reaction. I had the privilege of reading an unreleased copy to write this foreword, and I understood why it created such a stir. While it's a technical book, calling it that does it a disservice. Technical books often intimidate. They're stuffed with big words, obscure terms, convoluted examples that, unintentionally, make you feel stupid. The more experienced the author, the easier it is to forget what it's like to learn new concepts, to be a beginner.

Despite their decades of programming experience, Dave and Andy have conquered the difficult challenge of writing with the same excitement of people who've just learned these lessons. They don't talk down to you. They don't assume you are an expert. They don't even assume you've read the first edition. They take you as you are—programmers who just want to be better. They spend the pages of this book helping you get there, one actionable step at a time.

To be fair, they'd already done this before. The original release was full of tangible examples, new ideas, and practical tips to build your coding muscles and develop your coding brain that still apply today. But this updated edition makes two improvements on the book.

The first is the obvious one: it removes some of the older references, the out-of-date examples, and replaces them with fresh, modern content. You won't find examples of loop invariants or build machines. Dave and Andy have taken their powerful content and made sure the lessons still come through, free of the distractions of old examples. It dusts off old ideas like DRY (don't repeat yourself) and gives them a fresh coat of paint, really making them shine.

But the second is what makes this release truly exciting. After writing the first edition, they had the chance to reflect on what they were trying to say, what they wanted their readers to take away, and how it was being received. They got feedback on those lessons. They saw what stuck, what needed refining, what was misunderstood. In the twenty years that this book has made its way through the hands and hearts of programmers all over the world, Dave and Andy have studied this response and formulated new ideas, new concepts.

They've learned the importance of agency and recognized that developers have arguably more agency than most other professionals. They start this book with the simple but profound message: "it's your life." It reminds us of our own power in our code base, in our jobs, in our careers. It sets the tone for everything else in the book—that it's more than just another technical book filled with code examples.

What makes it truly stand out among the shelves of technical books is that it understands what it means to be a programmer. Programming is about trying to make the future less painful. It's about making things easier for our teammates. It's about getting things wrong and being able to bounce back. It's about forming good habits. It's about understanding your toolset. Coding is just part of the world of being a programmer, and this book explores that world.

I spend a lot of time thinking about the coding journey. I didn't grow up coding; I didn't study it in college. I didn't spend my teenage years tinkering with tech. I entered the coding world in my mid-twenties and had to learn what it meant to be a programmer. This community is very different from others I'd been a part of. There is a unique dedication to learning and practicality that is both refreshing and intimidating.

For me, it really does feel like entering a new world. A new town, at least. I had to get to know the neighbors, pick my grocery store, find the best coffee shops. It took a while to get the lay of the land, to find the most efficient routes, to avoid the streets with the heaviest traffic, to know when traffic was likely to hit. The weather is different, I needed a new wardrobe.

The first few weeks, even months, in a new town can be scary. Wouldn't it be wonderful to have a friendly, knowledgeable neighbor who'd been living there a while? Who can give you a tour, show you those coffee shops? Someone who'd been there long enough to know the culture, understand the pulse of the town, so you not only feel at home, but become a contributing member as well? Dave and Andy are those neighbors.

As a relative newcomer, it's easy to be overwhelmed not by the act of programming but the process of becoming a programmer. There is an entire mindset shift that needs to happen—a change in habits, behaviors, and expectations. The process of becoming a better programmer doesn't just happen because you know how to code; it must be met with intention and deliberate practice. This book is a guide to becoming a better programmer efficiently.

But make no mistake—it doesn't tell you how programming should be. It's not philosophical or judgmental in that way. It tells you, plain and simple, what a Pragmatic Programmer is—how they operate, and how they approach code. They leave it up to you to decide if you want to be one. If you feel it's not for you, they won't hold it against you. But if you decide it is, they're your friendly neighbors, there to show you the way.

▶ *Saron Yitbarek*

Founder & CEO of CodeNewbie
Host of Command Line Heroes

Preface to the Second Edition

Back in the 1990s, we worked with companies whose projects were having problems. We found ourselves saying the same things to each: maybe you should test that before you ship it; why does the code only build on Mary's machine? Why didn't anyone ask the users?

To save time with new clients, we started jotting down notes. And those notes became *The Pragmatic Programmer*. To our surprise the book seemed to strike a chord, and it has continued to be popular these last 20 years.

But 20 years is many lifetimes in terms of software. Take a developer from 1999 and drop them into a team today, and they'd struggle in this strange new world. But the world of the 1990s is equally foreign to today's developer. The book's references to things such as CORBA, CASE tools, and indexed loops were at best quaint and more likely confusing.

At the same time, 20 years has had no impact whatsoever on common sense. Technology may have changed, but people haven't. Practices and approaches that were a good idea then remain a good idea now. Those aspects of the book aged well.

So when it came time to create this *20th Anniversary Edition*, we had to make a decision. We could go through and update the technologies we reference and call it a day. Or we could reexamine the assumptions behind the practices we recommended in the light of an additional two decades' worth of experience.

In the end, we did both.

As a result, this book is something of a *Ship of Theseus*.[1] Roughly one-third of the topics in the book are brand new. Of the rest, the majority have been rewritten, either partially or totally. Our intent was to make things clearer, more relevant, and hopefully somewhat timeless.

1. If, over the years, every component of a ship is replaced as it fails, is the resulting vessel the same ship?

We made some difficult decisions. We dropped the *Resources* appendix, both because it would be impossible to keep up-to-date and because it's easier to search for what you want. We reorganized and rewrote topics to do with concurrency, given the current abundance of parallel hardware and the dearth of good ways of dealing with it. We added content to reflect changing attitudes and environments, from the agile movement which we helped launch, to the rising acceptance of functional programming idioms and the growing need to consider privacy and security.

Interestingly, though, there was considerably less debate between us on the content of this edition than there was when we wrote the first. We both felt that the stuff that was important was easier to identify.

Anyway, this book is the result. Please enjoy it. Maybe adopt some new practices. Maybe decide that some of the stuff we suggest is wrong. Get involved in your craft. Give us feedback.

But, most important, remember to make it fun.

How the Book Is Organized

This book is written as a collection of short topics. Each topic is self-contained, and addresses a particular theme. You'll find numerous cross references, which help put each topic in context. Feel free to read the topics 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 xxi). 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.

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 an appendix, but very few have a single *correct* solution. The challenges might form the basis of group discussions or essay work in advanced programming courses.

There's also a short bibliography listing the books and articles we explicitly reference.

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 email 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 website.[2]

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

2. https://pragprog.com/titles/tpp20

Send Us Feedback

We'd appreciate hearing from you. Email us at ppbook@pragprog.com.

Second Edition Acknowledgments

We have enjoyed literally thousands of interesting conversations about programming over the last 20 years, meeting people at conferences, at courses, and sometimes even on the plane. Each one of these has added to our understanding of the development process, and has contributed to the updates in this edition. Thank you all (and keep telling us when we're wrong).

Thanks to the participants in the book's beta process. Your questions and comments helped us explain things better.

Before we went beta, we shared the book with a few folks for comments. Thanks to VM (Vicky) Brasseur, Jeff Langr, and Kim Shrier for your detailed comments, and to José Valim and Nick Cuthbert for your technical reviews.

Thanks to Ron Jeffries for letting us use the Sudoku example.

Much gratitude to the folks at Pearson who agreed to let us create this book our way.

A special thanks to the indispensable Janet Furlow, who masters whatever she takes on and keeps us in line.

And, finally, a shout out to all the Pragmatic Programmers out there who have been making programming better for everyone for the last twenty years. Here's to twenty more.

From the Preface to the First Edition

This book will help you become a better programmer.

You could be a lone developer, a member of a large project team, or a consultant working with many clients at once. It doesn't matter; 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 in turn is derived from the Greek πραγματικός, meaning "fit for use."

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 *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 their 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 something 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 quantum computing 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. Deeply understanding 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:

Tip 1 Care About Your Craft

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

Tip 2 Think! About Your Work

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 project. 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 is an engineering discipline," they say, "that breaks down if individual team members make decisions for themselves."

We strongly disagree.

There *should* be engineering in software construction. 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.*

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 how your skills have grown.

CHAPTER 1

A Pragmatic Philosophy

This book is about you.

Make no mistake, it is *your* career, and more importantly, *It's Your Life*. You own it. You're here because you know you can become a better developer and help others become better as well. You can become a *Pragmatic Programmer*.

What distinguishes Pragmatic Programmers? We feel it's an attitude, a style, a philosophy of approaching problems and their solutions. They think beyond the immediate problem, placing it in its larger context and seeking out the bigger picture. After all, without this larger context, how can you be pragmatic? How can you make intelligent compromises and informed decisions?

Another key to their success is that Pragmatic Programmers take responsibility for everything they do, which we discuss in *The Cat Ate My Source Code*. Being responsible, Pragmatic Programmers won't sit idly by and watch their projects fall apart through neglect. In *Software Entropy*, we tell you how to keep your projects pristine.

Most people find change difficult, sometimes for good reasons, sometimes because of plain old inertia. In *Stone Soup and Boiled Frogs*, we look at a strategy for instigating change and (in the interests of balance) present the cautionary tale of an amphibian that ignored the dangers of gradual change.

One of the benefits of understanding the context in which you work is that it becomes easier to know just how good your software has to be. Sometimes near-perfection is the only option, but often there are trade-offs involved. We explore this in *Good-Enough Software*.

Of course, you need to have a broad base of knowledge and experience to pull all of this off. Learning is a continuous and ongoing process. In *Your Knowledge Portfolio*, we discuss some strategies for keeping the momentum up.

Finally, none of us works in a vacuum. We all spend a large amount of time interacting with others. *Communicate!* lists ways we can do this better.

Pragmatic programming stems from a philosophy of pragmatic thinking. This chapter sets the basis for that philosophy.

1 It's Your Life

I'm not in this world to live up to your expectations and you're not in this world to live up to mine.

➤ *Bruce Lee*

It is *your* life. You own it. You run it. You create it.

Many developers we talk to are frustrated. Their concerns are varied. Some feel they're stagnating in their job, others that technology has passed them by. Folks feel they are under appreciated, or underpaid, or that their teams are toxic. Maybe they want to move to Asia, or Europe, or work from home.

And the answer we give is always the same.

"Why can't you change it?"

Software development must appear close to the top of any list of careers where you have control. Our skills are in demand, our knowledge crosses geographic boundaries, we can work remotely. We're paid well. We really can do just about anything we want.

But, for some reason, developers seem to resist change. They hunker down, and hope things will get better. They look on, passively, as their skills become dated and complain that their companies don't train them. They look at ads for exotic locations on the bus, then step off into the chilling rain and trudge into work.

So here's the most important tip in the book.

Tip 3 You Have Agency

Does your work environment suck? Is your job boring? Try to fix it. But don't try forever. As Martin Fowler says, "you can change your organization or change your organization."[1]

1. http://wiki.c2.com/?ChangeYourOrganization

If technology seems to be passing you by, make time (in your own time) to study new stuff that looks interesting. You're investing in yourself, so doing it while you're off-the-clock is only reasonable.

Want to work remotely? Have you asked? If they say no, then find someone who says yes.

This industry gives you a remarkable set of opportunities. Be proactive, and take them.

Related Sections Include

- Topic 4, *Stone Soup and Boiled Frogs*, on page 8
- Topic 6, *Your Knowledge Portfolio*, on page 13

2

The Cat Ate My Source Code

The greatest of all weaknesses is the fear of appearing weak.

➤ *J.B. Bossuet, Politics from Holy Writ, 1709*

One of the cornerstones of the pragmatic philosophy is the idea of taking responsibility for yourself and your actions in terms of your career advancement, your learning and education, your project, and your day-to-day work. Pragmatic Programmers take charge of their own career, and aren't afraid to admit ignorance or error. It's not the most pleasant aspect of programming, to be sure, but it will happen—even on the best of projects. Despite thorough testing, good documentation, and solid automation, things go wrong. Deliveries are late. Unforeseen technical problems come up.

These things happen, and we try to deal with them as professionally as we can. This means being honest and direct. We can be proud of our abilities, but we must own up to our shortcomings—our ignorance and our mistakes.

Team Trust

Above all, your team needs to be able to trust and rely on you—and you need to be comfortable relying on each of them as well. Trust in a team is absolutely essential for creativity and collaboration according to the research literature.[2] In a healthy environment based in trust, you can safely speak your mind,

2. See, for example, a good meta-analysis at *Trust and team performance: A meta-analysis of main effects, moderators, and covariates*, http://dx.doi.org/10.1037/apl0000110

present your ideas, and rely on your team members who can in turn rely on you. Without trust, well…

Imagine a high-tech, stealth ninja team infiltrating the villain's evil lair. After months of planning and delicate execution, you've made it on site. Now it's your turn to set up the laser guidance grid: "Sorry, folks, I don't have the laser. The cat was playing with the red dot and I left it at home."

That sort of breach of trust might be hard to repair.

Take Responsibility

Responsibility is something you actively agree to. You make a commitment to ensure that something is done right, but you don't necessarily have direct control over every aspect of it. In addition to doing your own personal best, you must analyze the situation for risks that are beyond your control. You have the right *not* to take on a responsibility for an impossible situation, or one in which the risks are too great, or the ethical implications too sketchy. You'll have to make the call based on your own values and judgment.

When you *do* accept the responsibility for an outcome, you should expect to be held accountable for it. When you make a mistake (as we all do) or an error in judgment, admit it honestly and try to offer options.

Don't blame someone or something else, or make up an excuse. Don't blame all the problems on a vendor, a programming language, management, or your coworkers. Any and all of these may play a role, but it is up to *you* to provide solutions, not excuses.

If there was a risk that the vendor wouldn't come through for you, then you should have had a contingency plan. If your mass storage melts—taking all of your source code with it—and you don't have a backup, it's your fault. Telling your boss "the cat ate my source code" just won't cut it.

Tip 4 Provide Options, Don't Make Lame Excuses

Before you approach anyone to tell them why something can't be done, is late, or is broken, stop and listen to yourself. Talk to the rubber duck on your monitor, or the cat. Does your excuse sound reasonable, or stupid? How's it going to sound to your boss?

Run through the conversation in your mind. What is the other person likely to say? Will they ask, "Have you tried this…" or "Didn't you consider that?" How will you respond? Before you go and tell them the bad news, is there

anything else you can try? Sometimes, you just *know* what they are going to say, so save them the trouble.

Instead of excuses, provide options. Don't say it can't be done; explain what *can* be done to salvage the situation. Does code have to be deleted? Tell them so, and explain the value of refactoring (see Topic 40, *Refactoring*, on page 209).

Do you need to spend time prototyping to determine the best way to proceed (see Topic 13, *Prototypes and Post-it Notes*, on page 56)? Do you need to introduce better testing (see Topic 41, *Test to Code*, on page 214, and *Ruthless and Continuous Testing*, on page 275) or automation to prevent it from happening again?

Perhaps you need additional resources to complete this task. Or maybe you need to spend more time with the users? Or maybe it's just you: do you need to learn some technique or technology in greater depth? Would a book or a course help? Don't be afraid to ask, or to admit that you need help.

Try to flush out the lame excuses before voicing them aloud. If you must, tell your cat first. After all, if little Tiddles is going to take the blame....

Related Sections Include

- Topic 49, *Pragmatic Teams*, on page 264

Challenges

- How do you react when someone—such as a bank teller, an auto mechanic, or a clerk—comes to you with a lame excuse? What do you think of them and their company as a result?
- When you find yourself saying, "I don't know," be sure to follow it up with "—but I'll find out." It's a great way to admit what you don't know, but then take responsibility like a pro.

Software Entropy

While software development is immune from almost all physical laws, the inexorable increase in *entropy* hits us hard. *Entropy* is a term from physics that refers to the amount of "disorder" in a system. Unfortunately, the laws of thermodynamics guarantee that the entropy in the universe tends toward a maximum. When disorder increases in software, we call it "software rot." Some folks might call it by the more optimistic term, "technical debt," with the implied notion that they'll pay it back someday. They probably won't.

Whatever the name, though, both debt and rot can spread uncontrollably.

There are many factors that can contribute to software rot. The most important one seems to be the psychology, or culture, at work on a project. Even if you are a team of one, your project's psychology can be a very delicate thing. Despite the best-laid plans and the best people, a project can still experience ruin and decay during its lifetime. Yet there are other projects that, despite enormous difficulties and constant setbacks, successfully fight nature's tendency toward disorder and manage to come out pretty well.

What makes the difference?

In inner cities, some buildings are beautiful and clean, while others are rotting hulks. Why? Researchers in the field of crime and urban decay discovered a fascinating trigger mechanism, one that very quickly turns a clean, intact, inhabited building into a smashed and abandoned derelict.[3]

A broken window.

One broken window, left unrepaired for any substantial length of time, instills in the inhabitants of the building a sense of abandonment—a sense that the powers that be don't care about the building. So another window gets broken. People start littering. Graffiti appears. Serious structural damage begins. In a relatively short span of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.

Why would that make a difference? Psychologists have done studies[4] that show hopelessness can be contagious. Think of the flu virus in close quarters. Ignoring a clearly broken situation reinforces the ideas that perhaps *nothing*

3. See *The police and neighborhood safety [WH82]*
4. See *Contagious depression: Existence, specificity to depressed symptoms, and the role of reassurance seeking [Joi94]*

can be fixed, that no one cares, all is doomed; all negative thoughts which can spread among team members, creating a vicious spiral.

Tip 5 Don't Live with Broken Windows

Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered. If there is insufficient time to fix it properly, then *board it up.* Perhaps you can comment out the offending code, or display a "Not Implemented" message, or substitute dummy data instead. Take *some* action to prevent further damage and to show that you're on top of the situation.

We've seen clean, functional systems deteriorate pretty quickly once windows start breaking. There are other factors that can contribute to software rot, and we'll touch on some of them elsewhere, but neglect *accelerates* the rot faster than any other factor.

You may be thinking that no one has the time to go around cleaning up all the broken glass of a project. If so, then you'd better plan on getting a dumpster, or moving to another neighborhood. Don't let entropy win.

First, Do No Harm

Andy once had an acquaintance who was obscenely rich. His house was immaculate, loaded with priceless antiques, *objets d'art*, and so on. One day, a tapestry that was hanging a little too close to a fireplace caught on fire. The fire department rushed in to save the day—and his house. But before they dragged their big, dirty hoses into the house, they stopped—with the fire raging—to roll out a mat between the front door and the source of the fire.

They didn't want to mess up the carpet.

Now that sounds pretty extreme. Surely the fire department's first priority is to put out the fire, collateral damage be damned. But they clearly had assessed the situation, were confident of their ability to manage the fire, and were careful not to inflict unnecessary damage to the property. That's the way it must be with software: don't cause collateral damage just because there's a crisis of some sort. One broken window is one too many.

One broken window—a badly designed piece of code, a poor management decision that the team must live with for the duration of the project—is all it takes to start the decline. If you find yourself working on a project with quite a few broken windows, it's all too easy to slip into the mindset of "All the rest

of this code is crap, I'll just follow suit." It doesn't matter if the project has been fine up to this point. In the original experiment leading to the "Broken Window Theory," an abandoned car sat for a week untouched. But once a single window was broken, the car was stripped and turned upside down within *hours*.

By the same token, if you find yourself on a project where the code is pristinely beautiful—cleanly written, well designed, and elegant—you will likely take extra special care not to mess it up, just like the firefighters. Even if there's a fire raging (deadline, release date, trade show demo, etc.), *you* don't want to be the first one to make a mess and inflict additional damage.

Just tell yourself, "No broken windows."

Related Sections Include

- Topic 10, *Orthogonality*, on page 39
- Topic 40, *Refactoring*, on page 209
- Topic 44, *Naming Things*, on page 238

Challenges

- Help strengthen your team by surveying your project neighborhood. Choose two or three broken windows and discuss with your colleagues what the problems are and what could be done to fix them.
- Can you tell when a window first gets broken? What is your reaction? If it was the result of someone else's decision, or a management edict, what can you do about it?

Stone Soup and Boiled Frogs

The three soldiers returning home from war were hungry. When they saw the village ahead their spirits lifted—they were sure the villagers would give them a meal. But when they got there, they found the doors locked and the windows closed. After many years of war, the villagers were short of food, and hoarded what they had.

Undeterred, the soldiers boiled a pot of water and carefully placed three stones into it. The amazed villagers came out to watch.

"This is stone soup," the soldiers explained. "Is that all you put in it?" asked the villagers. "Absolutely—although some say it tastes even better with a few carrots…" A villager ran off, returning in no time with a basket of carrots from his hoard.

A couple of minutes later, the villagers again asked "Is that it?"

"Well," said the soldiers, "a couple of potatoes give it body." Off ran another villager.

Over the next hour, the soldiers listed more ingredients that would enhance the soup: beef, leeks, salt, and herbs. Each time a different villager would run off to raid their personal stores.

Eventually they had produced a large pot of steaming soup. The soldiers removed the stones, and they sat down with the entire village to enjoy the first square meal any of them had eaten in months.

There are a couple of morals in the stone soup story. The villagers are tricked by the soldiers, who use the villagers' curiosity to get food from them. But more importantly, the soldiers act as a catalyst, bringing the village together so they can jointly produce something that they couldn't have done by themselves—a synergistic result. Eventually everyone wins.

Every now and then, you might want to emulate the soldiers.

You may be in a situation where you know exactly what needs doing and how to do it. The entire system just appears before your eyes—you know it's right. But ask permission to tackle the whole thing and you'll be met with delays and blank stares. People will form committees, budgets will need approval, and things will get complicated. Everyone will guard their own resources. Sometimes this is called "start-up fatigue."

It's time to bring out the stones. Work out what you *can* reasonably ask for. Develop it well. Once you've got it, show people, and let them marvel. Then say "of course, it *would* be better if we added…" Pretend it's not important. Sit back and wait for them to start asking you to add the functionality you originally wanted. People find it easier to join an ongoing success. Show them a glimpse of the future and you'll get them to rally around.[5]

Tip 6 Be a Catalyst for Change

5. While doing this, you may be comforted by the line attributed to Rear Admiral Dr. Grace Hopper: "It's easier to ask forgiveness than it is to get permission."

The Villagers' Side

On the other hand, the stone soup story is also about gentle and gradual deception. It's about focusing too tightly. The villagers think about the stones and forget about the rest of the world. We all fall for it, every day. Things just creep up on us.

We've all seen the symptoms. Projects slowly and inexorably get totally out of hand. Most software disasters start out too small to notice, and most project overruns happen a day at a time. Systems drift from their specifications feature by feature, while patch after patch gets added to a piece of code until there's nothing of the original left. It's often the accumulation of small things that breaks morale and teams.

Tip 7 Remember the Big Picture

We've never tried this—honest. But "they" say that if you take a frog and drop it into boiling water, it will jump straight back out again. However, if you place the frog in a pan of cold water, then gradually heat it, the frog won't notice the slow increase in temperature and will stay put until cooked.

Note that the frog's problem is different from the broken windows issue discussed in Topic 3, *Software Entropy*, on page 6. In the Broken Window Theory, people lose the will to fight entropy because they perceive that no one else cares. The frog just doesn't notice the change.

Don't be like the fabled frog. Keep an eye on the big picture. Constantly review what's happening around you, not just what you personally are doing.

Related Sections Include

- Topic 1, *It's Your Life*, on page 2
- Topic 38, *Programming by Coincidence*, on page 197

Challenges

- While reviewing a draft of the first edition, John Lakos raised the following issue: The soldiers progressively deceive the villagers, but the change they catalyze does them all good. However, by progressively deceiving the frog, you're doing it harm. Can you determine whether you're making stone soup or frog soup when you try to catalyze change? Is the decision subjective or objective?

- Quick, without looking, how many lights are in the ceiling above you? How many exits in the room? How many people? Is there anything out of context, anything that looks like it doesn't belong? This is an exercise in *situational awareness*, a technique practiced by folks ranging from Boy and Girl Scouts to Navy SEALs. Get in the habit of really looking and noticing your surroundings. Then do the same for your project.

Good-Enough Software

Striving to better, oft we mar what's well.

➤ *Shakespeare, King Lear 1.4*

There's an old(ish) joke about a company that places an order for 100,000 ICs with a Japanese manufacturer. Part of the specification was the defect rate: one chip in 10,000. A few weeks later the order arrived: one large box containing thousands of ICs, and a small one containing just ten. Attached to the small box was a label that read: "These are the faulty ones."

If only we really had this kind of control over quality. But the real world just won't let us produce much that's truly perfect, particularly not bug-free software. Time, technology, and temperament all conspire against us.

However, this doesn't have to be frustrating. As Ed Yourdon described in an article in *IEEE Software*, *When good-enough software is best [You95]*, you can discipline yourself to write software that's good enough—good enough for your users, for future maintainers, for your own peace of mind. You'll find that you are more productive and your users are happier. And you may well find that your programs are actually better for their shorter incubation.

Before we go any further, we need to qualify what we're about to say. The phrase "good enough" does not imply sloppy or poorly produced code. All systems must meet their users' requirements to be successful, and meet basic performance, privacy, and security standards. We are simply advocating that users be given an opportunity to participate in the process of deciding when what you've produced is good enough for their needs.

Involve Your Users in the Trade-Off

Normally you're writing software for other people. Often you'll remember to find out what they want.[6] But do you ever ask them *how good* they want their

6. That was supposed to be a joke!

software to be? Sometimes there'll be no choice. If you're working on pacemakers, an autopilot, or a low-level library that will be widely disseminated, the requirements will be more stringent and your options more limited.

However, if you're working on a brand-new product, you'll have different constraints. The marketing people will have promises to keep, the eventual end users may have made plans based on a delivery schedule, and your company will certainly have cash-flow constraints. It would be unprofessional to ignore these users' requirements simply to add new features to the program, or to polish up the code just one more time. We're not advocating panic: it is equally unprofessional to promise impossible time scales and to cut basic engineering corners to meet a deadline.

The scope and quality of the system you produce should be discussed as part of that system's requirements.

Tip 8 Make Quality a Requirements Issue

Often you'll be in situations where trade-offs are involved. Surprisingly, many users would rather use software with some rough edges *today* than wait a year for the shiny, bells-and-whistles version (and in fact what they will need a year from now may be completely different anyway). Many IT departments with tight budgets would agree. Great software today is often preferable to the fantasy of perfect software tomorrow. If you give your users something to play with early, their feedback will often lead you to a better eventual solution (see Topic 12, *Tracer Bullets*, on page 50).

Know When to Stop

In some ways, programming is like painting. You start with a blank canvas and certain basic raw materials. You use a combination of science, art, and craft to determine what to do with them. You sketch out an overall shape, paint the underlying environment, then fill in the details. You constantly step back with a critical eye to view what you've done. Every now and then you'll throw a canvas away and start again.

But artists will tell you that all the hard work is ruined if you don't know when to stop. If you add layer upon layer, detail over detail, *the painting becomes lost in the paint.*

Don't spoil a perfectly good program by overembellishment and overrefinement. Move on, and let your code stand in its own right for a while. It may not be perfect. Don't worry: it could never be perfect. (In Chapter 7, *While You Are*