The Pragmatic Programmer

from journeyman
to master

Andrew Hunt
David Thomas

The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt
David Thomas

ADDISON–WESLEY
An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts Harlow, England Menlo Park, California
Berkeley, California Don Mills, Ontario Sydney
Bonn Amsterdam Tokyo Mexico City

AWL Direct Sales
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867
(781) 944-3700

Visit AWL on the Web: www.awl.com/cseng

Library of Congress Cataloging-in-Publication Data

Hunt, Andrew, 1964–
The Pragmatic Programmer / Andrew Hunt, David Thomas.
p. cm.
Includes bibliographical references.
ISBN 0-201-61622-X
1. Computer programming. I. Thomas, David, 1956– .
II. Title.
QA76.6.H857 1999
005.1--dc21 99–43581
CIP

ISBN 0-201-61622-X
Text printed on recycled and acid-free paper.
1 2 3 4 5 6 7 8 9 10—CRS—0302010099
Eighteenth printing, August 2006

Contents

Appendices

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:

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 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

Kirby, John Lakos, Pete McBreen, Carey P. Morris, Jared Richardson, Kevin Ruland, Eric Starr, Eric Vought, Chris Van Wyk, and Deborra Zukowski. Without their careful comments and valuable insights, this book would be less readable, less accurate, and twice as long. Thank you all for your time and wisdom.

The second printing of this book benefited greatly from the eagle eyes of our readers. Many thanks to Brian Blank, Paul Boal, Tom Ekberg, Brent Fulgham, Louis Paul Hebert, Henk-Jan Olde Loohuis, Alan Lund, Gareth McCaughan, Yoshiki Shibata, and Volker Wurst, both for finding the mistakes and for having the grace to point them out gently.

Over the years, we have worked with a large number of progressive clients, where we gained and refined the experience we write about here. Recently, we've been fortunate to work with Peter Gehrke on several large projects. His support and enthusiasm for our techniques are much appreciated.

This book was produced using LaTeX, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract, and SmallEiffel, using the Bash and zsh shells under Linux. The staggering thing is that all of this tremendous software is freely available. We owe a huge "thank you" to the thousands of Pragmatic Programmers worldwide who have contributed these and other works to us all. We'd particularly like to thank Reto Kramer for his help with iContract.

Last, but in no way least, we owe a huge debt to our families. Not only have they put up with late night typing, huge telephone bills, and our permanent air of distraction, but they've had the grace to read what we've written, time after time. Thank you for letting us dream.

Andy Hunt
Dave Thomas

Chapter 1

A Pragmatic Philosophy

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, always trying to place it in its larger context, always trying to be aware of 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 they 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 to accept, 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.

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 project, and your day-to-day work. A Pragmatic Programmer takes charge of his or her own career, and isn'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 be honest about our shortcomings—our ignorance as well as our mistakes.

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. You'll have to make the call based on your own ethics 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 the disk crashes—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 3

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 thrown out? Educate them on the value of refactoring (see *Refactoring*, page 184). Do you need to spend time prototyping to determine the best way to proceed (see *Prototypes and Post-it Notes*, page 53)? Do you need to introduce better testing (see *Code That's Easy to Test*, page 189, and *Ruthless Testing*, page 237) or automation (see *Ubiquitous Automation*, page 230) to prevent it from happening again? Perhaps you need additional resources. 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:

- *Prototypes and Post-it Notes*, page 53
- *Refactoring*, page 184
- *Code That's Easy to Test*, page 189
- *Ubiquitous Automation*, page 230
- *Ruthless Testing*, page 237

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?

Software Entropy

While software development is immune from almost all physical laws, *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, programmers call it "software rot."

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 [WK82].

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 space of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.

The "Broken Window Theory" has inspired police departments in New York and other major cities to crack down on the small stuff in order to keep out the big stuff. It works: keeping on top of broken windows, graffiti, and other small infractions has reduced the serious crime level.

TIP 4

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 you continue to think like that, then you'd better plan on getting a dumpster, or moving to another neighborhood. Don't let entropy win.

Putting Out Fires

By contrast, there's the story of an obscenely rich acquaintance of Andy's. His house was immaculate, beautiful, loaded with priceless antiques, *objets d'art*, and so on. One day, a tapestry that was hanging a little too close to his living room 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.

A pretty extreme case, to be sure, but that's the way it must be with software. 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 team and 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.

Related sections include:

- *Stone Soup and Boiled Frogs*, page 7
- *Refactoring*, page 184
- *Pragmatic Teams*, page 224

Challenges

- Help strengthen your team by surveying your computing "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?

3

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.[1]

TIP 5

Be a Catalyst for Change

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 6

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.

---

1. 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."

Note that the frog's problem is different from the broken windows issue discussed in Section 2. 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 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:

- *Software Entropy*, page 4
- *Programming by Coincidence*, page 172
- *Refactoring*, page 184
- *The Requirements Pit*, page 202
- *Pragmatic Teams*, page 224

Challenges

- While reviewing a draft of this book, 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?

Good-Enough Software

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

► ***King Lear* 1.4**

There's an old(ish) joke about a U.S. company that places an order for 100,000 integrated circuits 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* [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. 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.

Involve Your Users in the Trade-Off

Normally you're writing software for other people. Often you'll remember to get requirements from them.[2] But how often do you ask them *how good* they want their software to be? Sometimes there'll be no choice. If you're working on pacemakers, the space shuttle, 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.

2. That was supposed to be a joke!

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

TIP 7

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 multimedia version. Many IT departments with tight budgets would agree. Great software today is often preferable to 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 *Tracer Bullets*, page 48).

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 over-refinement. 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 6, page 171, we'll discuss philosophies for developing code in an imperfect world.)

Related sections include:

- *Tracer Bullets*, page 48
- *The Requirements Pit*, page 202
- *Pragmatic Teams*, page 224
- *Great Expectations*, page 255

Challenges

- Look at the manufacturers of the software tools and operating systems that you use. Can you find any evidence that these companies are comfortable shipping software they know is not perfect? As a user, would you rather (1) wait for them to get all the bugs out, (2) have complex software and accept some bugs, or (3) opt for simpler software with fewer defects?

- Consider the effect of modularization on the delivery of software. Will it take more or less time to get a monolithic block of software to the required quality compared with a system designed in modules? Can you find commercial examples?

5 Your Knowledge Portfolio

An investment in knowledge always pays the best interest.

► **Benjamin Franklin**

Ah, good old Ben Franklin—never at a loss for a pithy homily. Why, if we could just be early to bed and early to rise, we'd be great programmers—right? The early bird might get the worm, but what happens to the early worm?

In this case, though, Ben really hit the nail on the head. Your knowledge and experience are your most important professional assets.

Unfortunately, they're *expiring assets*.[3] Your knowledge becomes out of date as new techniques, languages, and environments are developed. Changing market forces may render your experience obsolete or irrelevant. Given the speed at which Web-years fly by, this can happen pretty quickly.

As the value of your knowledge declines, so does your value to your company or client. We want to prevent this from ever happening.

---

3. An *expiring asset* is something whose value diminishes over time. Examples include a warehouse full of bananas and a ticket to a ball game.

Your Knowledge Portfolio

We like to think of all the facts programmers know about computing, the application domains they work in, and all their experience as their *Knowledge Portfolios*. Managing a knowledge portfolio is very similar to managing a financial portfolio:

1. Serious investors invest regularly—as a habit.
2. Diversification is the key to long-term success.
3. Smart investors balance their portfolios between conservative and high-risk, high-reward investments.
4. Investors try to buy low and sell high for maximum return.
5. Portfolios should be reviewed and rebalanced periodically.

To be successful in your career, you must manage your knowledge portfolio using these same guidelines.

Building Your Portfolio

- **Invest regularly.** Just as in financial investing, you must invest in your knowledge portfolio *regularly*. Even if it's just a small amount, the habit itself is as important as the sums. A few sample goals are listed in the next section.
- **Diversify.** The more *different* things you know, the more valuable you are. As a baseline, you need to know the ins and outs of the particular technology you are working with currently. But don't stop there. The face of computing changes rapidly—hot technology today may well be close to useless (or at least not in demand) tomorrow. The more technologies you are comfortable with, the better you will be able to adjust to change.
- **Manage risk.** Technology exists along a spectrum from risky, potentially high-reward to low-risk, low-reward standards. It's not a good idea to invest all of your money in high-risk stocks that might collapse suddenly, nor should you invest all of it conservatively and miss out on possible opportunities. Don't put all your technical eggs in one basket.

- **Buy low, sell high.** Learning an emerging technology before it becomes popular can be just as hard as finding an undervalued stock, but the payoff can be just as rewarding. Learning Java when it first came out may have been risky, but it paid off handsomely for the early adopters who are now at the top of that field.

- **Review and rebalance.** This is a very dynamic industry. That hot technology you started investigating last month might be stone cold by now. Maybe you need to brush up on that database technology that you haven't used in a while. Or perhaps you could be better positioned for that new job opening if you tried out that other language. . . .

Of all these guidelines, the most important one is the simplest to do:

TIP 8

Invest Regularly in Your Knowledge Portfolio

Goals

Now that you have some guidelines on what and when to add to your knowledge portfolio, what's the best way to go about acquiring intellectual capital with which to fund your portfolio? Here are a few suggestions.

- **Learn at least one new language every year.** Different languages solve the same problems in different ways. By learning several different approaches, you can help broaden your thinking and avoid getting stuck in a rut. Additionally, learning many languages is far easier now, thanks to the wealth of freely available software on the Internet (see page 267).

- **Read a technical book each quarter.** Bookstores are full of technical books on interesting topics related to your current project. Once you're in the habit, read a book a month. After you've mastered the technologies you're currently using, branch out and study some that *don't* relate to your project.

- **Read nontechnical books, too.** It is important to remember that computers are used by *people*—people whose needs you are trying to satisfy. Don't forget the human side of the equation.

- **Take classes.** Look for interesting courses at your local community college or university, or perhaps at the next trade show that comes to town.

- **Participate in local user groups.** Don't just go and listen, but actively participate. Isolation can be deadly to your career; find out what people are working on outside of your company.

- **Experiment with different environments.** If you've worked only in Windows, play with Unix at home (the freely available Linux is perfect for this). If you've used only `makefiles` and an editor, try an IDE, and vice versa.

- **Stay current.** Subscribe to trade magazines and other journals (see page 262 for recommendations). Choose some that cover technology different from that of your current project.

- **Get wired.** Want to know the ins and outs of a new language or other technology? Newsgroups are a great way to find out what experiences other people are having with it, the particular jargon they use, and so on. Surf the Web for papers, commercial sites, and any other sources of information you can find.

It's important to continue investing. Once you feel comfortable with some new language or bit of technology, move on. Learn another one.

It doesn't matter whether you ever use any of these technologies on a project, or even whether you put them on your resume. The process of learning will expand your thinking, opening you to new possibilities and new ways of doing things. The cross-pollination of ideas is important; try to apply the lessons you've learned to your current project. Even if your project doesn't use that technology, perhaps you can borrow some ideas. Get familiar with object orientation, for instance, and you'll write plain C programs differently.

Opportunities for Learning

So you're reading voraciously, you're on top of all the latest breaking developments in your field (not an easy thing to do), and somebody asks you a question. You don't have the faintest idea what the answer is, and freely admit as much.

Don't let it stop there. Take it as a personal challenge to find the answer. Ask a guru. (If you don't have a guru in your office, you should be able to find one on the Internet: see the box on on the facing page.) Search the Web. Go to the library.[4]

If you can't find the answer yourself, find out who *can*. Don't let it rest. Talking to other people will help build your personal network, and you may surprise yourself by finding solutions to other, unrelated problems along the way. And that old portfolio just keeps getting bigger. . . .

All of this reading and researching takes time, and time is already in short supply. So you need to plan ahead. Always have something to read in an otherwise dead moment. Time spent waiting for doctors and dentists can be a great opportunity to catch up on your reading—but be sure to bring your own magazine with you, or you might find yourself thumbing through a dog-eared 1973 article about Papua New Guinea.

Critical Thinking

The last important point is to think *critically* about what you read and hear. You need to ensure that the knowledge in your portfolio is accurate and unswayed by either vendor or media hype. Beware of the zealots who insist that their dogma provides the *only* answer—it may or may not be applicable to you and your project.

Never underestimate the power of commercialism. Just because a Web search engine lists a hit first doesn't mean that it's the best match; the content provider can pay to get top billing. Just because a bookstore features a book prominently doesn't mean it's a good book, or even popular; they may have been paid to place it there.

TIP 9

Critically Analyze What You Read and Hear

Unfortunately, there are very few simple answers anymore. But with your extensive portfolio, and by applying some critical analysis to the

4. In this era of the Web, many people seem to have forgotten about real live libraries filled with research material and staff.

Care and Cultivation of Gurus

With the global adoption of the Internet, gurus suddenly are as close as your `Enter` key. So, how do you find one, and how do you get one to talk with you?

We find there are some simple tricks.

- Know exactly what you want to ask, and be as specific as you can be.
- Frame your question carefully and politely. Remember that you're asking a favor; don't seem to be demanding an answer.
- Once you've framed your question, stop and look again for the answer. Pick out some keywords and search the Web. Look for appropriate FAQs (lists of frequently asked questions with answers).
- Decide if you want to ask publicly or privately. Usenet newsgroups are wonderful meeting places for experts on just about any topic, but some people are wary of these groups' public nature. Alternatively, you can always e-mail your guru directly. Either way, use a meaningful subject line. ("`Need Help!!!`" doesn't cut it.)
- Sit back and be patient. People are busy, and it may take days to get a specific answer.

Finally, please be sure to thank anyone who responds to you. And if you see people asking questions *you* can answer, play your part and participate.

torrent of technical publications you will read, you can understand the *complex* answers.

Challenges

- Start learning a new language this week. Always programmed in C++? Try Smalltalk [URL 13] or Squeak [URL 14]. Doing Java? Try Eiffel [URL 10] or TOM [URL 15]. See page 267 for sources of other free compilers and environments.
- Start reading a new book (but finish this one first!). If you are doing very detailed implementation and coding, read a book on design and architecture. If you are doing high-level design, read a book on coding techniques.

- Get out and talk technology with people who aren't involved in your current project, or who don't work for the same company. Network in your company cafeteria, or maybe seek out fellow enthusiasts at a local user's group meeting.

6

Communicate!

I believe that it is better to be looked over than it is to be overlooked.

► **Mae West, *Belle of the Nineties*, 1934**

Maybe we can learn a lesson from Ms. West. It's not just what you've got, but also how you package it. Having the best ideas, the finest code, or the most pragmatic thinking is ultimately sterile unless you can communicate with other people. A good idea is an orphan without effective communication.

As developers, we have to communicate on many levels. We spend hours in meetings, listening and talking. We work with end users, trying to understand their needs. We write code, which communicates our intentions to a machine and documents our thinking for future generations of developers. We write proposals and memos requesting and justifying resources, reporting our status, and suggesting new approaches. And we work daily within our teams to advocate our ideas, modify existing practices, and suggest new ones. A large part of our day is spent communicating, so we need to do it well.

We've put together a list of ideas that we find useful.

Know What You Want to Say

Probably the most difficult part of the more formal styles of communication used in business is working out exactly what it is you want to say. Fiction writers plot out their books in detail before they start, but people writing technical documents are often happy to sit down at a keyboard, enter "1. Introduction," and start typing whatever comes into their heads next.

Plan what you want to say. Write an outline. Then ask yourself, "Does this get across whatever I'm trying to say?" Refine it until it does.

This approach is not just applicable to writing documents. When you're faced with an important meeting or a phone call with a major client, jot down the ideas you want to communicate, and plan a couple of strategies for getting them across.

Know Your Audience

You're communicating only if you're conveying information. To do that, you need to understand the needs, interests, and capabilities of your audience. We've all sat in meetings where a development geek glazes over the eyes of the vice president of marketing with a long monologue on the merits of some arcane technology. This isn't communicating: it's just talking, and it's annoying.[5]

Form a strong mental picture of your audience. The acrostic WISDOM, shown in Figure 1.1 on the following page, may help.

Say you want to suggest a Web-based system to allow your end users to submit bug reports. You can present this system in many different ways, depending on your audience. End users will appreciate that they can submit bug reports 24 hours a day without waiting on the phone. Your marketing department will be able to use this fact to boost sales. Managers in the support department will have two reasons to be happy: fewer staff will be needed, and problem reporting will be automated. Finally, developers may enjoy getting experience with Web-based client-server technologies and a new database engine. By making the appropriate pitch to each group, you'll get them all excited about your project.

Choose Your Moment

It's six o'clock on Friday afternoon, following a week when the auditors have been in. Your boss's youngest is in the hospital, it's pouring rain outside, and the commute home is guaranteed to be a nightmare. This probably isn't a good time to ask her for a memory upgrade for your PC.

As part of understanding what your audience needs to hear, you need to work out what their priorities are. Catch a manager who's just been given a hard time by her boss because some source code got lost, and

5. The word *annoy* comes from the Old French *enui*, which also means "to bore."

Figure 1.1. The WISDOM acrostic—understanding an audience

What do you want them to learn?
What is their **i**nterest in what you've got to say?
How **s**ophisticated are they?
How much **d**etail do they want?
Whom do you want to **o**wn the information?
How can you **m**otivate them to listen to you?

you'll have a more receptive listener to your ideas on source code repositories. Make what you're saying relevant in time, as well as in content. Sometimes all it takes is the simple question "Is this a good time to talk about...?"

Choose a Style

Adjust the style of your delivery to suit your audience. Some people want a formal "just the facts" briefing. Others like a long, wide-ranging chat before getting down to business. When it comes to written documents, some like to receive large bound reports, while others expect a simple memo or e-mail. If in doubt, ask.

Remember, however, that you are half of the communication transaction. If someone says they need a paragraph describing something and you can't see any way of doing it in less than several pages, tell them so. Remember, that kind of feedback is a form of communication, too.

Make It Look Good

Your ideas are important. They deserve a good-looking vehicle to convey them to your audience.

Too many developers (and their managers) concentrate solely on content when producing written documents. We think this is a mistake. Any chef will tell you that you can slave in the kitchen for hours only to ruin your efforts with poor presentation.

There is no excuse today for producing poor-looking printed documents. Modern word processors (along with layout systems such as LaTeX and troff) can produce stunning output. You need to learn just a few basic commands. If your word processor supports style sheets, use

them. (Your company may already have defined style sheets that you can use.) Learn how to set page headers and footers. Look at the sample documents included with your package to get ideas on style and layout. *Check the spelling,* first automatically and then by hand. After awl, their are spelling miss steaks that the chequer can knot ketch.

Involve Your Audience

We often find that the documents we produce end up being less important than the process we go through to produce them. If possible, involve your readers with early drafts of your document. Get their feedback, and pick their brains. You'll build a good working relationship, and you'll probably produce a better document in the process.

Be a Listener

There's one technique that you must use if you want people to listen to you: *listen to them.* Even if this is a situation where you have all the information, even if this is a formal meeting with you standing in front of 20 suits—if you don't listen to them, they won't listen to you.

Encourage people to talk by asking questions, or have them summarize what you tell them. Turn the meeting into a dialog, and you'll make your point more effectively. Who knows, you might even learn something.

Get Back to People

If you ask someone a question, you feel they're impolite if they don't respond. But how often do you fail to get back to people when they send you an e-mail or a memo asking for information or requesting some action? In the rush of everyday life, it's easy to forget. Always respond to e-mails and voice mails, even if the response is simply "I'll get back to you later." Keeping people informed makes them far more forgiving of the occasional slip, and makes them feel that you haven't forgotten them.

TIP 10

It's Both What You Say and the Way You Say It

Unless you work in a vacuum, you need to be able to communicate. The more effective that communication, the more influential you become.

E-Mail Communication

Everything we've said about communicating in writing applies equally to electronic mail. E-mail has evolved to the point where it is a mainstay of intra- and intercorporate communications. E-mail is used to discuss contracts, to settle disputes, and as evidence in court. But for some reason, people who would never send out a shabby paper document are happy to fling nasty-looking e-mail around the world.

Our e-mail tips are simple:

- Proofread before you hit SEND.
- Check the spelling.
- Keep the format simple. Some people read e-mail using proportional fonts, so the ASCII art pictures you laboriously created will look to them like hen-scratchings.
- Use rich-text or HTML formatted mail only if you know that all your recipients can read it. Plain text is universal.
- Try to keep quoting to a minimum. No one likes to receive back their own 100-line e-mail with "I agree" tacked on.
- If you're quoting other people's e-mail, be sure to attribute it, and quote it inline (rather than as an attachment).
- Don't flame unless you want it to come back and haunt you later.
- Check your list of recipients before sending. A recent *Wall Street Journal* article described an employee who took to distributing criticisms of his boss over departmental e-mail, without realizing that his boss was included on the distribution list.
- Archive and organize your e-mail—both the important stuff you receive and the mail you send.

As various Microsoft and Netscape employees discovered during the 1999 Department of Justice investigation, e-mail is forever. Try to give the same attention and care to e-mail as you would to any written memo or report.

Summary

- Know what you want to say.
- Know your audience.
- Choose your moment.
- Choose a style.
- Make it look good.
- Involve your audience.
- Be a listener.
- Get back to people.

Related sections include:

- *Prototypes and Post-it Notes*, page 53
- *Pragmatic Teams*, page 224

Challenges

- There are several good books that contain sections on communications within development teams [Bro95, McC95, DL99]. Make it a point to try to read all three over the next 18 months. In addition, the book *Dinosaur Brains* [Ber96] discusses the emotional baggage we all bring to the work environment.

- The next time you have to give a presentation, or write a memo advocating some position, try working through the WISDOM acrostic on page 20 before you start. See if it helps you understand how to position what you say. If appropriate, talk to your audience afterward and see how accurate your assessment of their needs was.

Finally, we all work in a world of limited time and resources. You can survive both of these scarcities better (and keep your bosses happier) if you get good at working out how long things will take, which we cover in *Estimating*.

By keeping these fundamental principles in mind during development, you can write code that's better, faster, and stronger. You can even make it look easy.

The Evils of Duplication

Giving a computer two contradictory pieces of knowledge was Captain James T. Kirk's preferred way of disabling a marauding artificial intelligence. Unfortunately, the same principle can be effective in bringing down *your* code.

As programmers, we collect, organize, maintain, and harness knowledge. We document knowledge in specifications, we make it come alive in running code, and we use it to provide the checks needed during testing.

Unfortunately, knowledge isn't stable. It changes—often rapidly. Your understanding of a requirement may change following a meeting with the client. The government changes a regulation and some business logic gets outdated. Tests may show that the chosen algorithm won't work. All this instability means that we spend a large part of our time in maintenance mode, reorganizing and reexpressing the knowledge in our systems.

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.

When we perform maintenance, we have to find and change the representations of things—those capsules of knowledge embedded in the application. The problem is that it's easy to duplicate knowledge in the specifications, processes, and programs that we develop, and when we do so, we invite a maintenance nightmare—one that starts well before the application ships.

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the *DRY* principle:

> EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS, AUTHORITATIVE REPRESENTATION WITHIN A SYSTEM.

Why do we call it *DRY*?

TIP 11

DRY—Don't Repeat Yourself

The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn't a question of whether you'll remember: it's a question of when you'll forget.

You'll find the *DRY* principle popping up time and time again throughout this book, often in contexts that have nothing to do with coding. We feel that it is one of the most important tools in the Pragmatic Programmer's tool box.

In this section we'll outline the problems of duplication and suggest general strategies for dealing with it.

How Does Duplication Arise?

Most of the duplication we see falls into one of the following categories:

- **Imposed duplication.** Developers feel they have no choice—the environment seems to require duplication.
- **Inadvertent duplication.** Developers don't realize that they are duplicating information.

- **Impatient duplication.** Developers get lazy and duplicate because it seems easier.
- **Interdeveloper duplication.** Multiple people on a team (or on different teams) duplicate a piece of information.

Let's look at these four *i*'s of duplication in more detail.

Imposed Duplication

Sometimes, duplication seems to be forced on us. Project standards may require documents that contain duplicated information, or documents that duplicate information in the code. Multiple target platforms each require their own programming languages, libraries, and development environments, which makes us duplicate shared definitions and procedures. Programming languages themselves require certain structures that duplicate information. We have all worked in situations where we felt powerless to avoid duplication. And yet often there are ways of keeping each piece of knowledge in one place, honoring the *DRY* principle, and making our lives easier at the same time. Here are some techniques:

Multiple representations of information. At the coding level, we often need to have the same information represented in different forms. Maybe we're writing a client-server application, using different languages on the client and server, and need to represent some shared structure on both. Perhaps we need a class whose attributes mirror the schema of a database table. Maybe you're writing a book and want to include excerpts of programs that you also will compile and test.

With a bit of ingenuity you can normally remove the need for duplication. Often the answer is to write a simple filter or code generator. Structures in multiple languages can be built from a common metadata representation using a simple code generator each time the software is built (an example of this is shown in Figure 3.4, page 106). Class definitions can be generated automatically from the online database schema, or from the metadata used to build the schema in the first place. The code extracts in this book are inserted by a preprocessor each time we format the text. The trick is to make the process active: this cannot be a one-time conversion, or we're back in a position of duplicating data.

Documentation in code. Programmers are taught to comment their code: good code has lots of comments. Unfortunately, they are never taught *why* code needs comments: bad code *requires* lots of comments.

The *DRY* principle tells us to keep the low-level knowledge in the code, where it belongs, and reserve the comments for other, high-level explanations. Otherwise, we're duplicating knowledge, and every change means changing both the code and the comments. The comments will inevitably become out of date, and untrustworthy comments are worse than no comments at all. (See *It's All Writing*, page 248, for more information on comments.)

Documentation and code. You write documentation, then you write code. Something changes, and you amend the documentation and update the code. The documentation and code both contain representations of the same knowledge. And we all know that in the heat of the moment, with deadlines looming and important clients clamoring, we tend to defer the updating of documentation.

Dave once worked on an international telex switch. Quite understandably, the client demanded an exhaustive test specification and required that the software pass all tests on each delivery. To ensure that the tests accurately reflected the specification, the team generated them programmatically from the document itself. When the client amended their specification, the test suite changed automatically. Once the team convinced the client that the procedure was sound, generating acceptance tests typically took only a few seconds.

Language issues. Many languages impose considerable duplication in the source. Often this comes about when the language separates a module's interface from its implementation. C and C++ have header files that duplicate the names and type information of exported variables, functions, and (for C++) classes. Object Pascal even duplicates this information in the same file. If you are using remote procedure calls or CORBA [URL 29], you'll duplicate interface information between the interface specification and the code that implements it.

There is no easy technique for overcoming the requirements of a language. While some development environments hide the need for header files by generating them automatically, and Object Pascal allows you to abbreviate repeated function declarations, you are generally stuck with

what you're given. At least with most language-based issues, a header file that disagrees with the implementation will generate some form of compilation or linkage error. You can still get things wrong, but at least you'll be told about it fairly early on.

Think also about comments in header and implementation files. There is absolutely no point in duplicating a function or class header comment between the two files. Use the header files to document interface issues, and the implementation files to document the nitty-gritty details that users of your code don't need to know.

Inadvertent Duplication

Sometimes, duplication comes about as the result of mistakes in the design.

Let's look at an example from the distribution industry. Say our analysis reveals that, among other attributes, a truck has a type, a license number, and a driver. Similarly, a delivery route is a combination of a route, a truck, and a driver. We code up some classes based on this understanding.

But what happens when Sally calls in sick and we have to change drivers? Both `Truck` and `DeliveryRoute` contain a driver. Which one do we change? Clearly this duplication is bad. Normalize it according to the underlying business model—does a truck really have a driver as part of its underlying attribute set? Does a route? Or maybe there needs to be a third object that knits together a driver, a truck, and a route. Whatever the eventual solution, avoid this kind of unnormalized data.

There is a slightly less obvious kind of unnormalized data that occurs when we have multiple data elements that are mutually dependent. Let's look at a class representing a line:

```
class Line {
 public:
  Point  start;
  Point  end;
  double length;
};
```

At first sight, this class might appear reasonable. A line clearly has a start and end, and will always have a length (even if it's zero). But we

newsgroups to allow developers to exchange ideas and ask questions. This provides a nonintrusive way of communicating—even across multiple sites—while retaining a permanent history of everything said.) Appoint a team member as the project librarian, whose job is to facilitate the exchange of knowledge. Have a central place in the source tree where utility routines and scripts can be deposited. And make a point of reading other people's source code and documentation, either informally or during code reviews. You're not snooping—you're learning from them. And remember, the access is reciprocal—don't get twisted about other people poring (pawing?) through *your* code, either.

TIP 12

Make It Easy to Reuse

What you're trying to do is foster an environment where it's easier to find and reuse existing stuff than to write it yourself. *If it isn't easy, people won't do it.* And if you fail to reuse, you risk duplicating knowledge.

Related sections include:

- *Orthogonality*, page 34
- *Text Manipulation*, page 99
- *Code Generators*, page 102
- *Refactoring*, page 184
- *Pragmatic Teams*, page 224
- *Ubiquitous Automation*, page 230
- *It's All Writing*, page 248

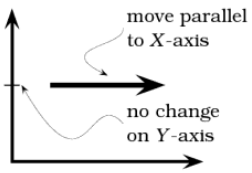
8

Orthogonality

Orthogonality is a critical concept if you want to produce systems that are easy to design, build, test, and extend. However, the concept of orthogonality is rarely taught directly. Often it is an implicit feature of various other methods and techniques you learn. This is a mistake. Once you learn to apply the principle of orthogonality directly, you'll notice an immediate improvement in the quality of systems you produce.

What Is Orthogonality?

"Orthogonality" is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are *independent*. Move along one of the lines, and your position projected onto the other doesn't change.



In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.

Before we look at the benefits of orthogonal systems, let's first look at a system that isn't orthogonal.

A Nonorthogonal System

You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch, suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground. You rationalize that the collective pitch lever[2] controls overall lift, so lower-

2. Helicopters have four basic controls. The *cyclic* is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the *collective pitch lever*. Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the *throttle*. Finally you have two foot *pedals*, which vary the amount of tail rotor thrust and so help turn the helicopter.

ing it slightly will start a gentle descent to the ground. However, when you try it, you discover that life isn't that simple. The helicopter's nose drops, and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.

Helicopter controls are decidedly not orthogonal.

Benefits of Orthogonality

As the helicopter example illustrates, nonorthogonal systems are inherently more complex to change and control. When components of any system are highly interdependent, there is no such thing as a local fix.

TIP 13

Eliminate Effects Between Unrelated Things

We want to design components that are self-contained: independent, and with a single, well-defined purpose (what Yourdon and Constantine call *cohesion* [YC86]). When components are isolated from one another, you know that you can change one without having to worry about the rest. As long as you don't change that component's external interfaces, you can be comfortable that you won't cause problems that ripple through the entire system.

You get two major benefits if you write orthogonal systems: increased productivity and reduced risk.

Gain Productivity

- Changes are localized, so development time and testing time are reduced. It is easier to write relatively small, self-contained components than a single large block of code. Simple components can be

designed, coded, unit tested, and then forgotten—there is no need to keep changing existing code as you add new code.

- An orthogonal approach also promotes reuse. If components have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned by their original implementors. The more loosely coupled your systems, the easier they are to reconfigure and reengineer.
- There is a fairly subtle gain in productivity when you combine orthogonal components. Assume that one component does M distinct things and another does N things. If they are orthogonal and you combine them, the result does $M \times N$ things. However, if the two components are not orthogonal, there will be overlap, and the result will do less. You get more functionality per unit effort by combining orthogonal components.

Reduce Risk

An orthogonal approach reduces the risks inherent in any development.

- Diseased sections of code are isolated. If a module is sick, it is less likely to spread the symptoms around the rest of the system. It is also easier to slice it out and transplant in something new and healthy.
- The resulting system is less fragile. Make small changes and fixes to a particular area, and any problems you generate will be restricted to that area.
- An orthogonal system will probably be better tested, because it will be easier to design and run tests on its components.
- You will not be as tightly tied to a particular vendor, product, or platform, because the interfaces to these third-party components will be isolated to smaller parts of the overall development.

Let's look at some of the ways you can apply the principle of orthogonality to your work.

Project Teams

Have you noticed how some project teams are efficient, with everyone knowing what to do and contributing fully, while the members of other

teams are constantly bickering and don't seem able to get out of each other's way?

Often this is an orthogonality issue. When teams are organized with lots of overlap, members are confused about responsibilities. Every change needs a meeting of the entire team, because any one of them *might* be affected.

How do you organize teams into groups with well-defined responsibilities and minimal overlap? There's no simple answer. It depends partly on the project and your analysis of the areas of potential change. It also depends on the people you have available. Our preference is to start by separating infrastructure from application. Each major infrastructure component (database, communications interface, middleware layer, and so on) gets its own subteam. Each obvious division of application functionality is similarly divided. Then we look at the people we have (or plan to have) and adjust the groupings accordingly.

You can get an informal measure of the orthogonality of a project team's structure. Simply see how many people *need* to be involved in discussing each change that is requested. The larger the number, the less orthogonal the group. Clearly, an orthogonal team is more efficient. (Having said this, we also encourage subteams to communicate constantly with each other.)

Design

Most developers are familiar with the need to design orthogonal systems, although they may use words such as *modular*, *component-based*, and *layered* to describe the process. Systems should be composed of a set of cooperating modules, each of which implements functionality independent of the others. Sometimes these components are organized into layers, each providing a level of abstraction. This layered approach is a powerful way to design orthogonal systems. Because each layer uses only the abstractions provided by the layers below it, you have great flexibility in changing underlying implementations without affecting code. Layering also reduces the risk of runaway dependencies between modules. You'll often see layering expressed in diagrams such as Figure 2.1 on the next page.

There is an easy test for orthogonal design. Once you have your components mapped out, ask yourself: *If I dramatically change the require-*

are normally generated by sprinkling explicit calls to some log function throughout your source. With AOP, you implement logging orthogonally to the things being logged. Using the Java version of AOP, you could write a log message when entering any method of class `Fred` by coding the *aspect*:

```
aspect Trace {
  advise * Fred.*(..) {
    static before {
      Log.write("-> Entering " + thisJoinPoint.methodName);
    }
  }
}
```

If you *weave* this aspect into your code, trace messages will be generated. If you don't, you'll see no messages. Either way, your original source is unchanged.

Coding

Every time you write code you run the risk of reducing the orthogonality of your application. Unless you constantly monitor not just what you are doing but also the larger context of the application, you might unintentionally duplicate functionality in some other module, or express existing knowledge twice.

There are several techniques you can use to maintain orthogonality:

- **Keep your code decoupled.** Write shy code—modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations. Try the Law of Demeter [LH89], which we discuss in *Decoupling and the Law of Demeter*, page 138. If you need to change an object's state, get the object to do it for you. This way your code remains isolated from the other code's implementation and increases the chances that you'll remain orthogonal.

- **Avoid global data.** Every time your code references global data, it ties itself into the other components that share that data. Even globals that you intend only to read can lead to trouble (for example, if you suddenly need to change your code to be multithreaded). In general, your code is easier to understand and maintain if you explicitly pass any required context into your modules. In object-oriented applications, context is often passed as parameters to

objects' constructors. In other code, you can create structures containing the context and pass around references to them.

The Singleton pattern in *Design Patterns* [GHJV95] is a way of ensuring that there is only one instance of an object of a particular class. Many people use these singleton objects as a kind of global variable (particularly in languages, such as Java, that otherwise do not support the concept of globals). Be careful with singletons—they can also lead to unnecessary linkage.

- **Avoid similar functions.** Often you'll come across a set of functions that all look similar—maybe they share common code at the start and end, but each has a different central algorithm. Duplicate code is a symptom of structural problems. Have a look at the Strategy pattern in *Design Patterns* for a better implementation.

Get into the habit of being constantly critical of your code. Look for any opportunities to reorganize it to improve its structure and orthogonality. This process is called *refactoring*, and it's so important that we've dedicated a section to it (see *Refactoring*, page 184).

Testing

An orthogonally designed and implemented system is easier to test. Because the interactions between the system's components are formalized and limited, more of the system testing can be performed at the individual module level. This is good news, because module level (or unit) testing is considerably easier to specify and perform than integration testing. In fact, we suggest that every module have its own unit test built into its code, and that these tests be performed automatically as part of the regular build process (see *Code That's Easy to Test*, page 189).

Building unit tests is itself an interesting test of orthogonality. What does it take to build and link a unit test? Do you have to drag in a large percentage of the rest of the system just to get a test to compile or link? If so, you've found a module that is not well decoupled from the rest of the system.

Bug fixing is also a good time to assess the orthogonality of the system as a whole. When you come across a problem, assess how localized