

Contents

FOREWORD	xiii
PREFACE	xvii
1 A PRAGMATIC PHILOSOPHY	1
1. The Cat Ate My Source Code	2
2. Software Entropy	4
3. Stone Soup and Boiled Frogs	7
4. Good-Enough Software	9
5. Your Knowledge Portfolio	12
6. Communicate!	18
2 A PRAGMATIC APPROACH	25
7. The Evils of Duplication	26
8. Orthogonality	34
9. Reversibility	44
10. Tracer Bullets	48
11. Prototypes and Post-it Notes	53
12. Domain Languages	57
13. Estimating	64
3 THE BASIC TOOLS	71
14. The Power of Plain Text	73
15. Shell Games	77
16. Power Editing	82
17. Source Code Control	86
18. Debugging	90
19. Text Manipulation	99
20. Code Generators	102

4	PRAGMATIC PARANOIA	107
21.	Design by Contract	109
22.	Dead Programs Tell No Lies	120
23.	Assertive Programming	122
24.	When to Use Exceptions	125
25.	How to Balance Resources	129
5	BEND, OR BREAK	137
26.	Decoupling and the Law of Demeter	138
27.	Metaprogramming	144
28.	Temporal Coupling	150
29.	It's Just a View	157
30.	Blackboards	165
6	WHILE YOU ARE CODING	171
31.	Programming by Coincidence	172
32.	Algorithm Speed	177
33.	Refactoring	184
34.	Code That's Easy to Test	189
35.	Evil Wizards	198
7	BEFORE THE PROJECT	201
36.	The Requirements Pit	202
37.	Solving Impossible Puzzles	212
38.	Not Until You're Ready	215
39.	The Specification Trap	217
40.	Circles and Arrows	220
8	PRAGMATIC PROJECTS	223
41.	Pragmatic Teams	224
42.	Ubiquitous Automation	230
43.	Ruthless Testing	237
44.	It's All Writing	248
45.	Great Expectations	255
46.	Pride and Prejudice	258

Appendices

A RESOURCES	261
Professional Societies	262
Building a Library	262
Internet Resources	266
Bibliography	275
B ANSWERS TO EXERCISES	279
INDEX	309

Sample pages

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 *πραττεω*, 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 3

The Basic Tools

Every craftsman starts his or her journey with a basic set of good-quality tools. A woodworker might need rules, gauges, a couple of saws, some good planes, fine chisels, drills and braces, mallets, and clamps. These tools will be lovingly chosen, will be built to last, will perform specific jobs with little overlap with other tools, and, perhaps most importantly, will feel right in the budding woodworker's hands.

Then begins a process of learning and adaptation. Each tool will have its own personality and quirks, and will need its own special handling. Each must be sharpened in a unique way, or held just so. Over time, each will wear according to use, until the grip looks like a mold of the woodworker's hands and the cutting surface aligns perfectly with the angle at which the tool is held. At this point, the tools become conduits from the craftsman's brain to the finished product—they have become extensions of his or her hands. Over time, the woodworker will add new tools, such as biscuit cutters, laser-guided miter saws, dovetail jigs—all wonderful pieces of technology. But you can bet that he or she will be happiest with one of those original tools in hand, feeling the plane sing as it slides through the wood.

Tools amplify your talent. The better your tools, and the better you know how to use them, the more productive you can be. Start with a basic set of generally applicable tools. As you gain experience, and as you come across special requirements, you'll add to this basic set. Like the craftsman, expect to add to your toolbox regularly. Always be on the lookout for better ways of doing things. If you come across a situation where you feel your current tools can't cut it, make a note to look for

something different or more powerful that would have helped. Let need drive your acquisitions.

Many new programmers make the mistake of adopting a single power tool, such as a particular integrated development environment (IDE), and never leave its cozy interface. This really is a mistake. We need to be comfortable beyond the limits imposed by an IDE. The only way to do this is to keep the basic tool set sharp and ready to use.

In this chapter we'll talk about investing in your own basic toolbox. As with any good discussion on tools, we'll start (in *The Power of Plain Text*) by looking at your raw materials, the stuff you'll be shaping. From there we'll move to the workbench, or in our case the computer. How can you use your computer to get the most out of the tools you use? We'll discuss this in *Shell Games*. Now that we have material and a bench to work on, we'll turn to the tool you'll probably use more than any other, your editor. In *Power Editing*, we'll suggest ways of making you more efficient.

To ensure that we never lose any of our precious work, we should always use a *Source Code Control* system—even for things such as our personal address book! And, since Mr. Murphy was really an optimist after all, you can't be a great programmer until you become highly skilled at *Debugging*.

You'll need some glue to bind much of the magic together. We discuss some possibilities, such as awk, Perl, and Python, in *Text Manipulation*.

Just as woodworkers sometimes build jigs to guide the construction of complex pieces, programmers can write code that itself writes code. We discuss this in *Code Generators*.

Spend time learning to use these tools, and at some point you'll be surprised to discover your fingers moving over the keyboard, manipulating text without conscious thought. The tools will have become extensions of your hands.

The Power of Plain Text

As Pragmatic Programmers, our base material isn't wood or iron, it's knowledge. We gather requirements as knowledge, and then express that knowledge in our designs, implementations, tests, and documents. And we believe that the best format for storing knowledge persistently is *plain text*. With plain text, we give ourselves the ability to manipulate knowledge, both manually and programmatically, using virtually every tool at our disposal.

What Is Plain Text?

Plain text is made up of printable characters in a form that can be read and understood directly by people. For example, although the following snippet is made up of printable characters, it is meaningless.

```
Field19=467abe
```

The reader has no idea what the significance of 467abe may be. A better choice would be to make it *understandable* to humans.

```
DrawingType=UMLActivityDrawing
```

Plain text doesn't mean that the text is unstructured; XML, SGML, and HTML are great examples of plain text that has a well-defined structure. You can do everything with plain text that you could do with some binary format, including versioning.

Plain text tends to be at a higher level than a straight binary encoding, which is usually derived directly from the implementation. Suppose you wanted to store a property called `uses_menus` that can be either `TRUE` or `FALSE`. Using text, you might write this as

```
myprop.uses_menus=FALSE
```

Contrast this with 0010010101110101.

The problem with most binary formats is that the context necessary to understand the data is separate from the data itself. You are artificially divorcing the data from its meaning. The data may as well be encrypted; it is absolutely meaningless without the application logic to parse it. With plain text, however, you can achieve a self-describing data stream that is independent of the application that created it.

TIP 20

Keep Knowledge in Plain Text

Drawbacks

There are two major drawbacks to using plain text: (1) It may take more space to store than a compressed binary format, and (2) it may be computationally more expensive to interpret and process a plain text file.

Depending on your application, either or both of these situations may be unacceptable—for example, when storing satellite telemetry data, or as the internal format of a relational database.

But even in these situations, it may be acceptable to store *metadata* about the raw data in plain text (see *Metaprogramming*, page 144).

Some developers may worry that by putting metadata in plain text, they're exposing it to the system's users. This fear is misplaced. Binary data may be more obscure than plain text, but it is no more secure. If you worry about users seeing passwords, encrypt them. If you don't want them changing configuration parameters, include a *secure hash*¹ of all the parameter values in the file as a checksum.

The Power of Text

Since *larger* and *slower* aren't the most frequently requested features from users, why bother with plain text? What *are* the benefits?

- Insurance against obsolescence
- Leverage
- Easier testing

Insurance Against Obsolescence

Human-readable forms of data, and self-describing data, will outlive all other forms of data and the applications that created them. Period.

1. MD5 is often used for this purpose. For an excellent introduction to the wonderful world of cryptography, see [Sch95].

As long as the data survives, you will have a chance to be able to use it—potentially long after the original application that wrote it is defunct.

You can parse such a file with only partial knowledge of its format; with most binary files, you must know all the details of the entire format in order to parse it successfully.

Consider a data file from some legacy system² that you are given. You know little about the original application; all that's important to you is that it maintained a list of clients' Social Security numbers, which you need to find and extract. Among the data, you see

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Recognizing the format of a Social Security number, you can quickly write a small program to extract that data—even if you have no information on anything else in the file.

But imagine if the file had been formatted this way instead:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

You may not have recognized the significance of the numbers quite as easily. This is the difference between *human readable* and *human understandable*.

While we're at it, FIELD10 doesn't help much either. Something like

```
<SSNO>123-45-6789</SSNO>
```

makes the exercise a no-brainer—and ensures that the data will outlive any project that created it.

Leverage

Virtually every tool in the computing universe, from source code management systems to compiler environments to editors and stand-alone filters, can operate on plain text.

2. All software becomes legacy as soon as it's written.

The Unix Philosophy

Unix is famous for being designed around the philosophy of small, sharp tools, each intended to do one thing well. This philosophy is enabled by using a common underlying format—the line-oriented, plain text file. Databases used for system administration (users and passwords, networking configuration, and so on) are all kept as plain text files. (Some systems, such as Solaris, also maintain a binary form of certain databases as a performance optimization. The plain text version is kept as an interface to the binary version.)

When a system crashes, you may be faced with only a minimal environment to restore it (you may not be able to access graphics drivers, for instance). Situations such as this can really make you appreciate the simplicity of plain text.

For instance, suppose you have a production deployment of a large application with a complex site-specific configuration file (sendmail comes to mind). If this file is in plain text, you could place it under a source code control system (see *Source Code Control*, page 86), so that you automatically keep a history of all changes. File comparison tools such as `diff` and `fc` allow you to see at a glance what changes have been made, while `sum` allows you to generate a checksum to monitor the file for accidental (or malicious) modification.

Easier Testing

If you use plain text to create synthetic data to drive system tests, then it is a simple matter to add, update, or modify the test data *without having to create any special tools to do so*. Similarly, plain text output from regression tests can be trivially analyzed (with `diff`, for instance) or subjected to more thorough scrutiny with Perl, Python, or some other scripting tool.

Lowest Common Denominator

Even in the future of XML-based intelligent agents that travel the wild and dangerous Internet autonomously, negotiating data interchange among themselves, the ubiquitous text file will still be there. In fact, in

heterogeneous environments the advantages of plain text can outweigh all of the drawbacks. You need to ensure that all parties can communicate using a common standard. Plain text is that standard.

Related sections include:

- *Source Code Control*, page 86
- *Code Generators*, page 102
- *Metaprogramming*, page 144
- *Blackboards*, page 165
- *Ubiquitous Automation*, page 230
- *It's All Writing*, page 248

Challenges

- Design a small address book database (name, phone number, and so on) using a straightforward binary representation in your language of choice. Do this before reading the rest of this challenge.
 1. Translate that format into a plain text format using XML.
 2. For each version, add a new, variable-length field called *directions* in which you might enter directions to each person's house.

What issues come up regarding versioning and extensibility? Which form was easier to modify? What about converting existing data?

Shell Games

Every woodworker needs a good, solid, reliable workbench, somewhere to hold work pieces at a convenient height while he or she works them. The workbench becomes the center of the wood shop, the craftsman returning to it time and time again as a piece takes shape.

For a programmer manipulating files of text, that workbench is the command shell. From the shell prompt, you can invoke your full repertoire of tools, using pipes to combine them in ways never dreamt of by their original developers. From the shell, you can launch applications, debuggers, browsers, editors, and utilities. You can search for files,

query the status of the system, and filter output. And by programming the shell, you can build complex macro commands for activities you perform often.

For programmers raised on GUI interfaces and integrated development environments (IDEs), this might seem an extreme position. After all, can't you do everything equally well by pointing and clicking?

The simple answer is “no.” GUI interfaces are wonderful, and they can be faster and more convenient for some simple operations. Moving files, reading MIME-encoded e-mail, and typing letters are all things that you might want to do in a graphical environment. But if you do all your work using GUIs, you are missing out on the full capabilities of your environment. You won't be able to automate common tasks, or use the full power of the tools available to you. And you won't be able to combine your tools to create customized *macro tools*. A benefit of GUIs is WYSIWYG—what you see is what you get. The disadvantage is WYSIAYG—what you see is *all* you get.

GUI environments are normally limited to the capabilities that their designers intended. If you need to go beyond the model the designer provided, you are usually out of luck—and more often than not, you *do* need to go beyond the model. Pragmatic Programmers don't just cut code, or develop object models, or write documentation, or automate the build process—we do *all* of these things. The scope of any one tool is usually limited to the tasks that the tool is expected to perform. For instance, suppose you need to integrate a code preprocessor (to implement design-by-contract, or multi-processing pragmas, or some such) into your IDE. Unless the designer of the IDE explicitly provided hooks for this capability, you can't do it.

You may already be comfortable working from the command prompt, in which case you can safely skip this section. Otherwise, you may need to be convinced that the shell is your friend.

As a Pragmatic Programmer, you will constantly want to perform ad hoc operations—things that the GUI may not support. The command line is better suited when you want to quickly combine a couple of commands to perform a query or some other task. Here are a few examples.

Find all .c files modified more recently than your Makefile.

Shell... `find . -name '*.c' -newer Makefile -print`

GUI..... Open the Explorer, navigate to the correct directory, click on the Makefile, and note the modification time. Then bring up Tools/Find, and enter *.c for the file specification. Select the date tab, and enter the date you noted for the Makefile in the first date field. Then hit OK.

Construct a zip/tar archive of my source.

Shell... `zip archive.zip *.h *.c - or -`
`tar cvf archive.tar *.h *.c`

GUI..... Bring up a ZIP utility (such as the shareware WinZip [URL 41]), select “Create New Archive,” enter its name, select the source directory in the add dialog, set the filter to “*.c”, click “Add,” set the filter to “*.h”, click “Add,” then close the archive.

Which Java files have not been changed in the last week?

Shell... `find . -name '*.java' -mtime +7 -print`

GUI..... Click and navigate to “Find files,” click the “Named” field and type in “*.java”, select the “Date Modified” tab. Then select “Between.” Click on the starting date and type in the starting date of the beginning of the project. Click on the ending date and type in the date of a week ago today (be sure to have a calendar handy). Click on “Find Now.”

Of those files, which use the awt libraries?

Shell... `find . -name '*.java' -mtime +7 -print |`
`xargs grep 'java.awt'`

GUI..... Load each file in the list from the previous example into an editor and search for the string “java.awt”. Write down the name of each file containing a match.

Clearly the list could go on. The shell commands may be obscure or terse, but they are powerful and concise. And, because shell commands can be combined into script files (or command files under Windows

systems), you can build sequences of commands to automate things you do often.

TIP 21

Use the Power of Command Shells

Gain familiarity with the shell, and you'll find your productivity soaring. Need to create a list of all the unique package names explicitly imported by your Java code? The following stores it in a file called "list."

```
grep '^import ' *.java |
  sed -e 's/.*import *//' -e 's/;.*$//' |
  sort -u >list
```

If you haven't spent much time exploring the capabilities of the command shell on the systems you use, this might appear daunting. However, invest some energy in becoming familiar with your shell and things will soon start falling into place. Play around with your command shell, and you'll be surprised at how much more productive it makes you.

Shell Utilities and Windows Systems

Although the command shells provided with Windows systems are improving gradually, Windows command-line utilities are still inferior to their Unix counterparts. However, all is not lost.

Cygnus Solutions has a package called Cygwin [URL 31]. As well as providing a Unix compatibility layer for Windows, Cygwin comes with a collection of more than 120 Unix utilities, including such favorites as `ls`, `grep`, and `find`. The utilities and libraries may be downloaded and used for free, but be sure to read their license.³ The Cygwin distribution comes with the Bash shell.

3. The GNU General Public License [URL 57] is a kind of legal virus that Open Source developers use to protect their (and your) rights. You should spend some time reading it. In essence, it says that you can use and modify GPL'd software, but if you distribute any modifications they must be licensed according to the GPL (and marked as such), and you must make source available. That's the virus part—whenever you derive a work from a GPL'd work, your derived work must also be GPL'd. However, it does not limit you in any way when simply using the tools—the ownership and licensing of software developed using the tools are up to you.

Using Unix Tools Under Windows

We love the availability of high-quality Unix tools under Windows, and use them daily. However, be aware that there are integration issues. Unlike their MS-DOS counterparts, these utilities are sensitive to the case of filenames, so `ls a*.bat` won't find `AUTOEXEC.BAT`. You may also come across problems with filenames containing spaces, and with differences in path separators. Finally, there are interesting problems when running MS-DOS programs that expect MS-DOS-style arguments under the Unix shells. For example, the Java utilities from JavaSoft use a colon as their `CLASSPATH` separator under Unix, but use a semicolon under MS-DOS. As a result, a Bash or ksh script that runs on a Unix box will run identically under Windows, but the command line it passes to Java will be interpreted incorrectly.

Alternatively, David Korn (of Korn shell fame) has put together a package called UWIN. This has the same aims as the Cygwin distribution—it is a Unix development environment under Windows. UWIN comes with a version of the Korn shell. Commercial versions are available from Global Technologies, Ltd. [URL 30]. In addition, AT&T allows free downloading of the package for evaluation and academic use. Again, read their license before using.

Finally, Tom Christiansen is (at the time of writing) putting together *Perl Power Tools*, an attempt to implement all the familiar Unix utilities portably, in Perl [URL 32].

Related sections include:

- *Ubiquitous Automation*, page 230

Challenges






- Are there things that you're currently doing manually in a GUI? Do you ever pass instructions to colleagues that involve a number of individual "click this button," "select this item" steps? Could these be automated?
- Whenever you move to a new environment, make a point of finding out what shells are available. See if you can bring your current shell with you.
- Investigate alternatives to your current shell. If you come across a problem your shell can't address, see if an alternative shell would cope better.

Power Editing

We've talked before about tools being an extension of your hand. Well, this applies to editors more than to any other software tool. You need to be able to manipulate text as effortlessly as possible, because text is the basic raw material of programming. Let's look at some common features and functions that help you get the most from your editing environment.

One Editor

We think it is better to know one editor very well, and use it for all editing tasks: code, documentation, memos, system administration, and so on. Without a single editor, you face a potential modern day Babel of confusion. You may have to use the built-in editor in each language's IDE for coding, and an all-in-one office product for documentation, and maybe a different built-in editor for sending e-mail. Even the keystrokes you use to edit command lines in the shell may be different.⁴ It is difficult to be proficient in any of these environments if you have a different set of editing conventions and commands in each.

You need to be proficient. Simply typing linearly and using a mouse to cut and paste is not enough. You just can't be as effective that way as you can with a powerful editor under your fingers. Typing  or  ten times to move the cursor left to the beginning of a line isn't as efficient as typing a single key such as , , or .

TIP 22

Use a Single Editor Well

Choose an editor, know it thoroughly, and use it for all editing tasks. If you use a single editor (or set of keybindings) across all text editing activities, you don't have to stop and think to accomplish text manipulation: the necessary keystrokes will be a reflex. The editor will be

4. Ideally, the shell you use should have keybindings that match the ones used by your editor. Bash, for instance, supports both vi and emacs keybindings.

an extension of your hand; the keys will sing as they slice their way through text and thought. That's our goal.

Make sure that the editor you choose is available on all platforms you use. Emacs, vi, CRISP, Brief, and others are available across multiple platforms, often in both GUI and non-GUI (text screen) versions.

Editor Features

Beyond whatever features you find particularly useful and comfortable, here are some basic abilities that we think every decent editor should have. If your editor falls short in any of these areas, then this may be the time to consider moving on to a more advanced one.

- **Configurable.** All aspects of the editor should be configurable to your preferences, including fonts, colors, window sizes, and key-stroke bindings (which keys perform what commands). Using only keystrokes for common editing operations is more efficient than mouse or menu-driven commands, because your hands never leave the keyboard.
- **Extensible.** An editor shouldn't be obsolete just because a new programming language comes out. It should be able to integrate with whatever compiler environment you are using. You should be able to "teach" it the nuances of any new language or text format (XML, HTML version 9, and so on).
- **Programmable.** You should be able to program the editor to perform complex, multistep tasks. This can be done with macros or with a built-in scripting programming language (Emacs uses a variant of Lisp, for instance).

In addition, many editors support features that are specific to a particular programming language, such as:

- Syntax highlighting
- Auto-completion
- Auto-indentation
- Initial code or document boilerplate
- Tie-in to help systems
- IDE-like features (compile, debug, and so on)

Figure 3.1. Sorting lines in an editor

<pre>import java.util.Vector; import java.util.Stack; import java.net.URL; import java.awt.*;</pre>	<pre>emacs: M-x sort-lines ↗ ↘ vi: :.,+3!sort</pre>	<pre>import java.awt.*; import java.net.URL; import java.util.Stack; import java.util.Vector;</pre>
---	---	---

A feature such as syntax highlighting may sound like a frivolous extra, but in reality it can be very useful and enhance your productivity. Once you get used to seeing keywords appear in a different color or font, a mistyped keyword that *doesn't* appear that way jumps out at you long before you fire up the compiler.

Having the ability to compile and navigate directly to errors within the editor environment is very handy on big projects. Emacs in particular is adept at this style of interaction.

Productivity

A surprising number of people we've met use the Windows notepad utility to edit their source code. This is like using a teaspoon as a shovel—simply typing and using basic mouse-based cut and paste is not enough.

What sort of things will you need to do that *can't* be done in this way?

Well, there's cursor movement, to start with. Single keystrokes that move you in units of words, lines, blocks, or functions are far more efficient than repeatedly typing a keystroke that moves you character by character or line by line.

Or suppose you are writing Java code. You like to keep your `import` statements in alphabetical order, and someone else has checked in a few files that don't adhere to this standard (this may sound extreme, but on a large project it can save you a lot of time scanning through a long list of `import` statements). You'd like to go quickly through a few files and sort a small section of them. In editors such as `vi` and Emacs you can do this easily (see Figure 3.1). Try *that* in notepad.

Some editors can help streamline common operations. For instance, when you create a new file in a particular language, the editor can supply a template for you. It might include:

- Name of the class or module filled in (derived from the filename)
- Your name and/or copyright statements
- Skeletons for constructs in that language (constructor and destructor declarations, for example)

Another useful feature is auto-indenting. Rather than having to indent manually (by using space or tab), the editor automatically indents for you at the appropriate time (after typing an open brace, for example). The nice part about this feature is that you can use the editor to provide a consistent indentation style for your project.⁵

Where to Go from Here

This sort of advice is particularly hard to write because virtually every reader is at a different level of comfort and expertise with the editor(s) they are currently using. So, to summarize, and to provide some guidance on where to go next, find yourself in the left-hand column of the chart, and look at the right-hand column to see what we think you should do.

<i>If this sounds like you...</i>	<i>Then think about...</i>
<i>I use only basic features of many different editors.</i>	Pick a powerful editor and learn it well.
<i>I have a favorite editor, but I don't use all of its features.</i>	Learn them. Cut down the number of keystrokes you need to type.
<i>I have a favorite editor and use it where possible.</i>	Try to expand and use it for more tasks than you do already.
<i>I think you are nuts. Notepad is the best editor ever made.</i>	As long as you are happy and productive, go for it! But if you find yourself subject to "editor envy," you may need to reevaluate your position.

5. The Linux kernel is developed this way. Here you have geographically dispersed developers, many working on the same pieces of code. There is a published list of settings (in this case, for Emacs) that describes the required indentation style.

What Editors Are Available?

Having recommended that you master a decent editor, which one do we recommend? Well, we're going to duck that question; your choice of editor is a personal one (some would even say a religious one!). However, in Appendix A, page 266, we list a number of popular editors and where to get them.

Challenges

- Some editors use full-blown languages for customization and scripting. Emacs, for example, uses Lisp. As one of the new languages you are going to learn this year, learn the language your editor uses. For anything you find yourself doing repeatedly, develop a set of macros (or equivalent) to handle it.
- Do you know everything your editor is capable of doing? Try to stump your colleagues who use the same editor. Try to accomplish any given editing task in as few keystrokes as possible.

Source Code Control

Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.

► **George Santayana, *Life of Reason***

One of the important things we look for in a user interface is the UNDO key—a single button that forgives us our mistakes. It's even better if the environment supports multiple levels of undo and redo, so you can go back and recover from something that happened a couple of minutes ago. But what if the mistake happened last week, and you've turned your computer on and off ten times since then? Well, that's one of the many benefits of using a source code control system: it's a giant UNDO key—a project-wide time machine that can return you to those halcyon days of last week, when the code actually compiled and ran.

Source code control systems, or the more widely scoped *configuration management* systems, keep track of every change you make in your source code and documentation. The better ones can keep track of