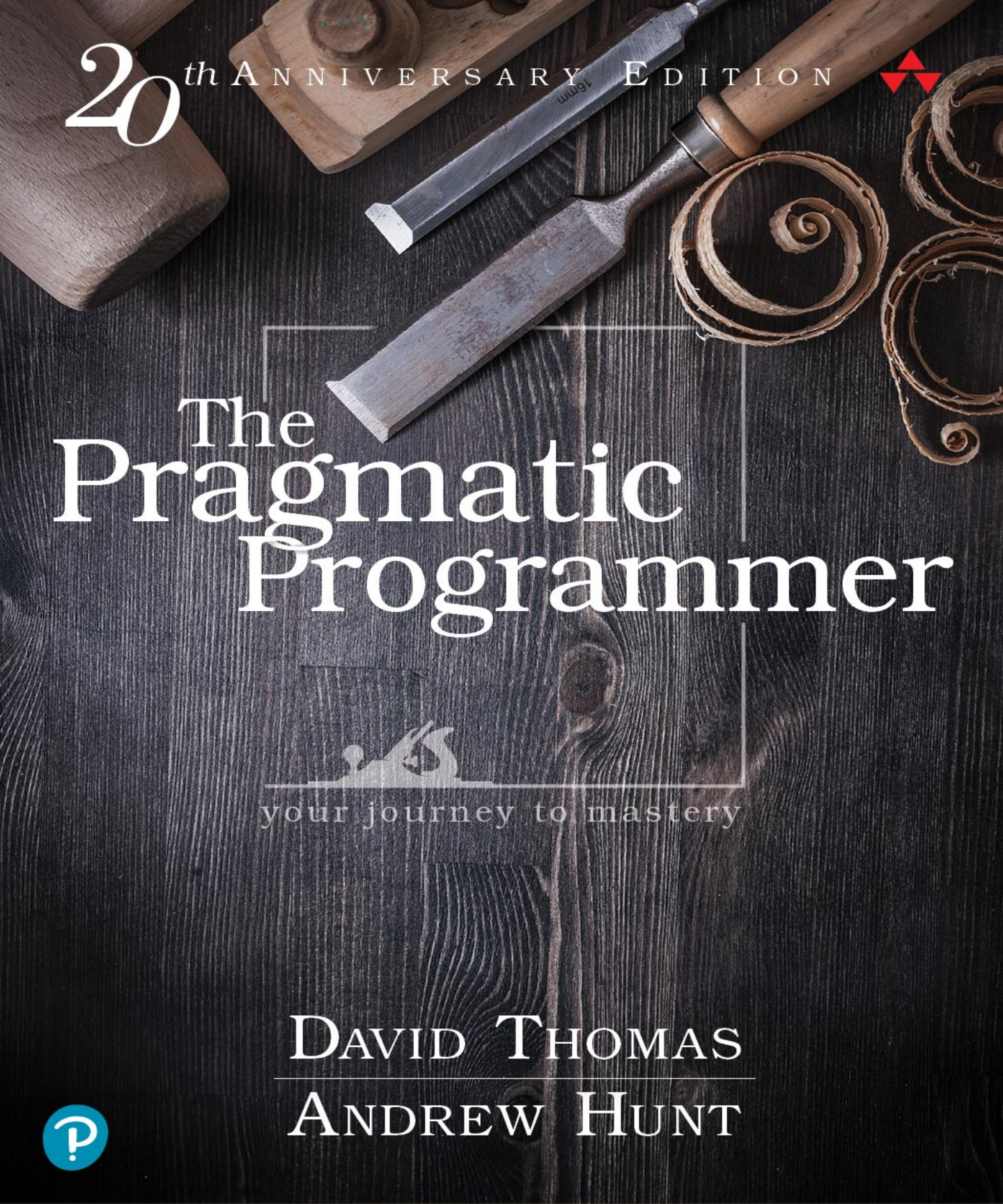
20th Anniversary Edition
The Pragmatic Programmer
your journey to mastery
David Thomas
Andrew Hunt

- Navigate to a particular line number.
- Sort selected lines.
- Search for both strings and regular expressions, and repeat previous searches.
- Temporarily create multiple cursors based on a selection or on a pattern match, and edit the text at each in parallel.
- Display compilation errors in the current project.
- Run the current project's tests.

Can you do all this without using a mouse/trackpad?

You might say that your current editor can't do some of these things. Maybe it's time to switch?

Moving Toward Fluency

We doubt there are more than a handful of people who know *all* the commands in any particular powerful editor. We don't expect you to, either. Instead, we suggest a more pragmatic approach: learn the commands that make your life easier.

The recipe for this is fairly simple.

First, look at yourself while you're editing. Every time you find yourself doing something repetitive, get into the habit of thinking "there must be a better way." Then find it.

Once you've discovered a new, useful feature, you now need to get it installed into your muscle memory, so you can use it without thinking. The only way we know to do that is through repetition. Consciously look for opportunities to use your new superpower, ideally many times a day. After a week or so, you'll find you use it without thinking.

Growing Your Editor

Most of the powerful code editors are built around a basic core that is then augmented through extensions. Many are supplied with the editor, and others can be added later.

When you bump into some apparent limitation of the editor you're using, search around for an extension that will do the job. The chances are that you are not alone in needing that capability, and if you're lucky someone else will have published their solution.

Take this a step further. Dig into your editor's extension language. Work out how to use it to automate some of the repetitive things you do. Often you'll just need a line or two of code.

Sometimes you might take it further still, and you'll find yourself writing a full-blown extension. If so, publish it: if you had a need for it, other people will, too.

Related Sections Include

- Topic 7, *Communicate!*, on page 19

Challenges

- No more autorepeat.

  Everyone does it: you need to delete the last word you typed, so you press down on `backspace` and wait for autorepeat to kick in. In fact, we bet that your brain has done this so much that you can judge pretty much exactly when to release the key.

  So turn off autorepeat, and instead learn the key sequences to move, select, and delete by characters, words, lines, and blocks.

- This one is going to hurt.

  Lose the mouse/trackpad. For one whole week, edit using just the keyboard. You'll discover a bunch of stuff that you can't do without pointing and clicking, so now's the time to learn. Keep notes (we recommend going old-school and using pencil and paper) of the key sequences you learn.

  You'll take a productivity hit for a few days. But, as you learn to do stuff without moving your hands away from the home position, you'll find that your editing becomes faster and more fluent than it ever was in the past.

- Look for integrations. While writing this chapter, Dave wondered if he could preview the final layout (a PDF file) in an editor buffer. One download later, the layout is sitting alongside the original text, all in the editor. Keep a list of things you'd like to bring into your editor, then look for them.

- Somewhat more ambitiously, if you can't find a plugin or extension that does what you want, write one. Andy is fond of making custom, local file-based Wiki plugins for his favorite editors. If you can't find it, build it!

19

Version 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 version control system (VCS): 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.

For many folks, that's the limit of their VCS usage. Those folks are missing out on a whole bigger world of collaboration, deployment pipelines, issue tracking, and general team interaction.

So let's take a look at VCS, first as a repository of changes, and then as a central meeting place for your team and their code.

Shared Directories Are *NOT* Version Control

We still come across the occasional team who share their project source files across a network: either internally or using some kind of cloud storage.

This is not viable.

Teams that do this are constantly messing up each other's work, losing changes, breaking builds, and getting into fist fights in the car park. It's like writing concurrent code with shared data and no synchronization mechanism. Use version control.

But there's more! Some folks *do* use version control, and keep their main repository on a network or cloud drive. They reason that this is the best of both worlds: their files are accessible anywhere and (in the case of cloud storage) it's backed up off-site.

Turns out that this is even worse, and you risk losing everything. The version control software uses a set of interacting files and directories. If two instances simultaneously make changes, the overall state can become corrupted, and there's no telling how much damage will be done. And no one likes seeing developers cry.

It Starts at the Source

Version control systems keep track of every change you make in your source code and documentation. With a properly configured source code control system, *you can always go back to a previous version of your software.*

But a version control system does far more than undo mistakes. A good VCS will let you track changes, answering questions such as: Who made changes in this line of code? What's the difference between the current version and last week's? How many lines of code did we change in this release? Which files get changed most often? This kind of information is invaluable for bug-tracking, audit, performance, and quality purposes.

A VCS will also let you identify releases of your software. Once identified, you will always be able to go back and regenerate the release, independent of changes that may have occurred later.

Version control systems may keep the files they maintain in a central repository—a great candidate for archiving.

Finally, version control systems allow two or more users to be working concurrently on the same set of files, even making concurrent changes in the same file. The system then manages the merging of these changes when the files are sent back to the repository. Although seemingly risky, such systems work well in practice on projects of all sizes.

Tip 28 Always Use Version Control

Always. Even if you are a single-person team on a one-week project. Even if it's a "throw-away" prototype. Even if the stuff you're working on isn't source code. Make sure that *everything* is under version control: documentation, phone number lists, memos to vendors, makefiles, build and release procedures, that little shell script that tidies up log files—everything. We routinely use version control on just about everything we type (including the text of this book). Even if we're not working on a project, our day-to-day work is secured in a repository.

Branching Out

Version control systems don't just keep a single history of your project. One of their most powerful and useful features is the way they let you isolate islands of development into things called *branches*. You can create a branch at any point in your project's history, and any work you do in that branch

will be isolated from all other branches. At some time in the future you can *merge* the branch you're working on back into another branch, so the target branch now contains the changes you made in your branch. Multiple people can even be working on a branch: in a way, branches are like little clone projects.

One benefit of branches is the isolation they give you. If you develop feature A in one branch, and a teammate works on feature B in another, you're not going to interfere with each other.

A second benefit, which may be surprising, is that branches are often at the heart of a team's project workflow.

And this is where things get a little confusing. Version control branches and test organization have something in common: they both have thousands of people out there telling you how you should do it. And that advice is largely meaningless, because what they're really saying is "this is what worked for me."

So use version control in your project, and if you bump into workflow issues, search for possible solutions. And remember to review and adjust what you're doing as you gain experience.

A Thought Experiment

Spill an entire cup of tea (English breakfast, with a little milk) onto your laptop keyboard. Take the machine to the smart-person bar, and have them tut and frown. Buy a new computer. Take it home.

How long would it take to get that machine back to the same state it was in (with all the SSH keys, editor configuration, shell setup, installed applications, and so on) at the point where you first lifted that fateful cup? This was an issue one of us faced recently.

Just about everything that defined the configuration and usage of the original machine was stored in version control, including:

- All the user preferences and dotfiles
- The editor configuration
- The list of software installed using Homebrew
- The Ansible script used to configure apps
- All current projects

The machine was restored by the end of the afternoon.

Version Control as a Project Hub

Although version control is incredibly useful on personal projects, it really comes into its own when working with a team. And much of this value comes from how you host your repository.

Now, many version control systems don't need any hosting. They are completely decentralized, with each developer cooperating on a peer-to-peer basis. But even with these systems, it's worth looking into having a central repository, because once you do, you can take advantage of a ton of integrations to make the project flow easier.

Many of the repository systems are open source, so you can install and run them in your company. But that's not really your line of business, so we'd recommend most people host with a third party. Look for features such as:

- Good security and access control
- Intuitive UI
- The ability to do everything from the command line, too (because you may need to automate it)
- Automated builds and tests
- Good support for branch merging (sometimes called pull requests)
- Issue management (ideally integrated into commits and merges, so you can keep metrics)
- Good reporting (a Kanban board-like display of pending issues and tasks can be very useful)
- Good team communications: emails or other notifications on changes, a wiki, and so on

Many teams have their VCS configured so that a push to a particular branch will automatically build the system, run the tests, and if successful deploy the new code into production.

Sound scary? Not when you realize you're using version control. You can always roll it back.

Related Sections Include

- Topic 11, *Reversibility*, on page 47
- Topic 49, *Pragmatic Teams*, on page 264
- Topic 51, *Pragmatic Starter Kit*, on page 273