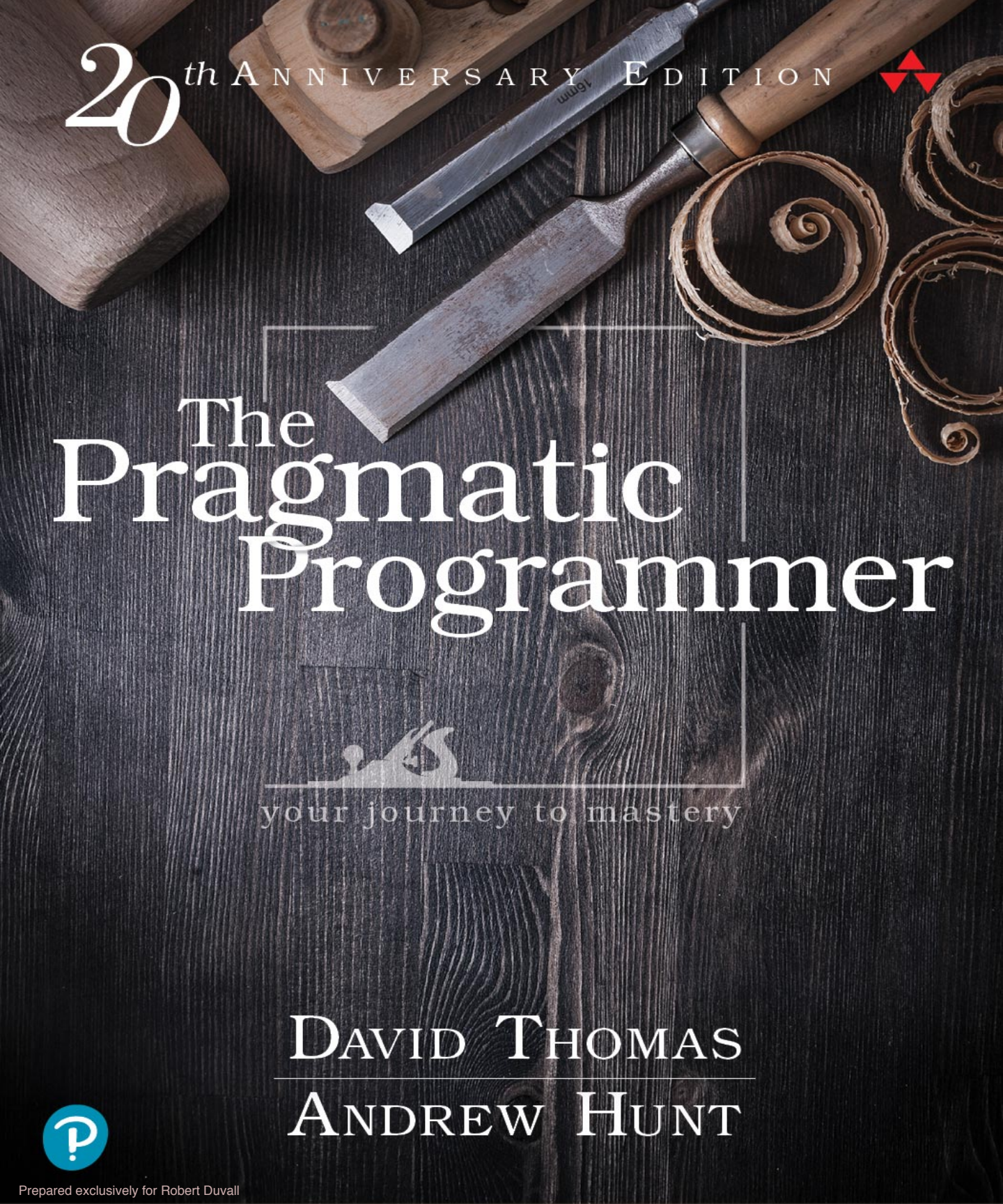

Challenges

- Knowing you can roll back to any previous state using the VCS is one thing, but can you actually do it? Do you know the commands to do it properly? Learn them now, not when disaster strikes and you're under pressure.
- Spend some time thinking about recovering your own laptop environment in case of a disaster. What would you need to recover? Many of the things you need are just text files. If they're not in a VCS (hosted off your laptop), find a way to add them. Then think about the other stuff: installed applications, system configuration, and so on. How can you express all that stuff in text files so it, too, can be saved?

  An interesting experiment, once you've made some progress, is to find an old computer you no longer use and see if your new system can be used to set it up.
- Consciously explore the features of your current VCS and hosting provider that you're not using. If your team isn't using feature branches, experiment with introducing them. The same with pull/merge requests. Continuous integration. Build pipelines. Even continuous deployment. Look into the team communication tools, too: wikis, Kanban boards, and the like.

  You don't have to use any of it. But you do need to know what it does so you can make that decision.
- Use version control for nonproject things, too.

Debugging

It is a painful thing
To look at your own trouble and know
That you yourself and no one else has made it

➤ *Sophocles, Ajax*

The word *bug* has been used to describe an "object of terror" ever since the fourteenth century. Rear Admiral Dr. Grace Hopper, the inventor of COBOL, is credited with observing the first *computer bug*—literally, a moth caught in a relay in an early computer system. When asked to explain why the machine wasn't behaving as intended, a technician reported that there was "a bug in the system," and dutifully taped it—wings and all—into the log book.

Regrettably, we still have bugs in the system, albeit not the flying kind. But the fourteenth century meaning—a bogeyman—is perhaps even more applicable now than it was then. Software defects manifest themselves in a variety of ways, from misunderstood requirements to coding errors. Unfortunately, modern computer systems are still limited to doing what you *tell* them to do, not necessarily what you *want* them to do.

No one writes perfect software, so it's a given that debugging will take up a major portion of your day. Let's look at some of the issues involved in debugging and some general strategies for finding elusive bugs.

Psychology of Debugging

Debugging is a sensitive, emotional subject for many developers. Instead of attacking it as a puzzle to be solved, you may encounter denial, finger pointing, lame excuses, or just plain apathy.

Embrace the fact that debugging is just *problem solving*, and attack it as such.

Having found someone else's bug, you can spend time and energy laying blame on the filthy culprit who created it. In some workplaces this is part of the culture, and may be cathartic. However, in the technical arena, you want to concentrate on fixing the *problem*, not the blame.

Tip 29 Fix the Problem, Not the Blame

It doesn't really matter whether the bug is your fault or someone else's. It is still your problem.

A Debugging Mindset

The easiest person to deceive is one's self.

➤ *Edward Bulwer-Lytton, The Disowned*

Before you start debugging, it's important to adopt the right mindset. You need to turn off many of the defenses you use each day to protect your ego, tune out any project pressures you may be under, and get yourself comfortable. Above all, remember the first rule of debugging:

Tip 30 Don't Panic

It's easy to get into a panic, especially if you are facing a deadline, or have a nervous boss or client breathing down your neck while you are trying to find the cause of the bug. But it is very important to step back a pace, and actually *think* about what could be causing the symptoms that you believe indicate a bug.

If your first reaction on witnessing a bug or seeing a bug report is "that's impossible," you are plainly wrong. Don't waste a single neuron on the train of thought that begins "but that can't happen" because quite clearly it *can*, and has.

Beware of myopia when debugging. Resist the urge to fix just the symptoms you see: it is more likely that the actual fault may be several steps removed from what you are observing, and may involve a number of other related things. Always try to discover the root cause of a problem, not just this particular appearance of it.

Where to Start

Before you *start* to look at the bug, make sure that you are working on code that built cleanly—without warnings. We routinely set compiler warning levels as high as possible. It doesn't make sense to waste time trying to find a problem that the computer could find for you! We need to concentrate on the harder problems at hand.

When trying to solve any problem, you need to gather all the relevant data. Unfortunately, bug reporting isn't an exact science. It's easy to be misled by coincidences, and you can't afford to waste time debugging coincidences. You first need to be accurate in your observations.

Accuracy in bug reports is further diminished when they come through a third party—you may actually need to *watch* the user who reported the bug in action to get a sufficient level of detail.

Andy once worked on a large graphics application. Nearing release, the testers reported that the application crashed every time they painted a stroke with a particular brush. The programmer responsible argued that there was nothing wrong with it; he had tried painting with it, and it worked just fine. This dialog went back and forth for several days, with tempers rapidly rising.

Finally, we got them together in the same room. The tester selected the brush tool and painted a stroke from the upper right corner to the lower left corner. The application exploded. "Oh," said the programmer, in a small voice, who

then sheepishly admitted that he had made test strokes only from the lower left to the upper right, which did not expose the bug.

There are two points to this story:

- You may need to interview the user who reported the bug in order to gather more data than you were initially given.
- Artificial tests (such as the programmer's single brush stroke from bottom to top) don't exercise enough of an application. You must brutally test both boundary conditions and realistic end-user usage patterns. You need to do this systematically (see *Ruthless and Continuous Testing*, on page 275).

Debugging Strategies

Once *you* think you know what is going on, it's time to find out what the *program* thinks is going on.

Reproducing Bugs

No, our bugs aren't really multiplying (although some of them are probably old enough to do it legally). We're talking about a different kind of reproduction.

The best way to start fixing a bug is to make it reproducible. After all, if you can't reproduce it, how will you know if it is ever fixed?

But we want more than a bug that can be reproduced by following some long series of steps; we want a bug that can be reproduced with a *single command*. It's a lot harder to fix a bug if you have to go through 15 steps to get to the point where the bug shows up.

So here's the most important rule of debugging:

Tip 31 Failing Test Before Fixing Code

Sometimes by forcing yourself to isolate the circumstances that display the bug, you'll even gain an insight on how to fix it. The act of writing the test informs the solution.

Coder in a Strange Land

All this talk about isolating the bug is fine, when faced with 50,000 lines of code and a ticking clock, what's a poor coder to do?

First, look at the problem. Is it a crash? It's always surprising when we teach courses that involve programming how many developers see an exception pop up in red and immediately tab across to the code.

> **Tip 32** Read the Damn Error Message

'nuf said.

Bad Results

What if it's not a crash? What if it's just a bad result?

Get in there with a debugger and use your failing test to trigger the problem.

Before anything else, make sure that you're also seeing the incorrect value in the debugger. We've both wasted hours trying to track down a bug only to discover that this particular run of the code worked fine.

Sometimes the problem is obvious: `interest_rate` is `4.5` and should be `0.045`. More often you have to look deeper to find out why the value is wrong in the first place. Make sure you know how to move up and down the call stack and examine the local stack environment.

We find it often helps to keep pen and paper nearby so we can jot down notes. In particular we often come across a clue and chase it down, only to find it didn't pan out. If we didn't jot down where we were when we started the chase, we could lose a lot of time getting back there.

Sometimes you're looking at a stack trace that seems to scroll on forever. In this case, there's often a quicker way to find the problem than examining each and every stack frame: use a *binary chop*. But before we discuss that, let's look at two other common bug scenarios.

Sensitivity to Input Values

You've been there. Your program works fine with all the test data, and survives its first week in production with honor. Then it suddenly crashes when fed a particular dataset.

You can try looking at the place it crashes and work backwards. But sometimes it's easier to start with the data. Get a copy of the dataset and feed it through a locally running copy of the app, making sure it still crashes. Then binary chop the data until you isolate exactly which input values are leading to the crash.

Regressions Across Releases

You're on a good team, and you release your software into production. At some point a bug pops up in code that worked OK a week ago. Wouldn't it be nice if you could identify the specific change that introduced it? Guess what? Binary chop time.

The Binary Chop

Every CS undergraduate has been forced to code a binary chop (sometimes called a binary search). The idea is simple. You're looking for a particular value in a sorted array. You could just look at each value in turn, but you'd end up looking at roughly half the entries on average until you either found the value you wanted, or you found a value greater than it, which would mean the value's not in the array.

But it's faster to use a *divide and conquer* approach. Choose a value in the middle of the array. If it's the one you're looking for, stop. Otherwise you can chop the array in two. If the value you find is greater than the target then you know it must be in the first half of the array, otherwise it's in the second half. Repeat the procedure in the appropriate subarray, and in no time you'll have a result. (As we'll see when we talk about *Big-O Notation*, on page 204, a linear search is $O(n)$, and a binary chop is $O(\log n)$).

So, the binary chop is way, way faster on any decent sized problem. Let's see how to apply it to debugging.

When you're facing a massive stacktrace and you're trying to find out exactly which function mangled the value in error, you do a chop by choosing a stack frame somewhere in the middle and seeing if the error is manifest there. If it is, then you know to focus on the frames before, otherwise the problem is in the frames after. Chop again. Even if you have 64 frames in the stacktrace, this approach will give you an answer after at most six attempts.

If you find bugs that appear on certain datasets, you might be able to do the same thing. Split the dataset into two, and see if the problem occurs if you feed one or the other through the app. Keep dividing the data until you get a minimum set of values that exhibit the problem.

If your team has introduced a bug during a set of releases, you can use the same type of technique. Create a test that causes the current release to fail. Then choose a half-way release between now and the last known working version. Run the test again, and decide how to narrow your search. Being able to do this is just one of the many benefits of having good version control in your projects. Indeed, many version control systems will take this further

and will automate the process, picking releases for you depending on the result of the test.

Logging and/or Tracing

Debuggers generally focus on the state of the program *now*. Sometimes you need more—you need to watch the state of a program or a data structure over time. Seeing a stack trace can only tell you how you got here directly. It typically can't tell you what you were doing prior to this call chain, especially in event-based systems.[2]

Tracing statements are those little diagnostic messages you print to the screen or to a file that say things such as "got here" and "value of x = 2." It's a primitive technique compared with IDE-style debuggers, but it is peculiarly effective at diagnosing several classes of errors that debuggers can't. Tracing is invaluable in any system where time itself is a factor: concurrent processes, real-time systems, and event-based applications.

You can use tracing statements to drill down into the code. That is, you can add tracing statements as you descend the call tree.

Trace messages should be in a regular, consistent format as you may want to parse them automatically. For instance, if you needed to track down a resource leak (such as unbalanced file opens/closes), you could trace each `open` and each `close` in a log file. By processing the log file with text processing tools or shell commands, you can easily identify where the offending `open` was occurring.

Rubber Ducking

A very simple but particularly useful technique for finding the cause of a problem is simply to explain it to someone else. The other person should look over your shoulder at the screen, and nod his or her head constantly (like a rubber duck bobbing up and down in a bathtub). They do not need to say a word; the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap off the screen and announce itself.[3]

It sounds simple, but in explaining the problem to another person you must explicitly state things that you may take for granted when going through the

2. Although the Elm language does have a time-traveling debugger.
3. Why "rubber ducking"? While an undergraduate at Imperial College in London, Dave did a lot of work with a research assistant named Greg Pugh, one of the best developers Dave has known. For several months Greg carried around a small yellow rubber duck, which he'd place on his terminal while coding. It was a while before Dave had the courage to ask....

code yourself. By having to verbalize some of these assumptions, you may suddenly gain new insight into the problem. And if you don't have a person, a rubber duck, or teddy bear, or potted plant will do.[4]

Process of Elimination

In most projects, the code you are debugging may be a mixture of application code written by you and others on your project team, third-party products (database, connectivity, web framework, specialized communications or algorithms, and so on) and the platform environment (operating system, system libraries, and compilers).

It is possible that a bug exists in the OS, the compiler, or a third-party product—but this should not be your first thought. It is much more likely that the bug exists in the application code under development. It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken. Even if the problem *does* lie with a third party, you'll still have to eliminate your code before submitting the bug report.

We worked on a project where a senior engineer was convinced that the select system call was broken on a Unix system. No amount of persuasion or logic could change his mind (the fact that every other networking application on the box worked fine was irrelevant). He spent weeks writing workarounds, which, for some odd reason, didn't seem to fix the problem. When finally forced to sit down and read the documentation on select, he discovered the problem and corrected it in a matter of minutes. We now use the phrase "select is broken" as a gentle reminder whenever one of us starts blaming the system for a fault that is likely to be our own.

Tip 33 "select" Isn't Broken

Remember, if you see hoof prints, think horses—not zebras. The OS is probably not broken. And select is probably just fine.

If you "changed only one thing" and the system stopped working, that one thing was likely to be responsible, directly or indirectly, no matter how far-fetched it seems. Sometimes the thing that changed is outside of your control: new versions of the OS, compiler, database, or other third-party software can wreak havoc with previously correct code. New bugs might show up. Bugs

4. Earlier versions of the book talked about talking to your *pot plant*. It was a typo. Honest.

for which you had a workaround get fixed, breaking the workaround. APIs change, functionality changes; in short, it's a whole new ball game, and you must retest the system under these new conditions. So keep a close eye on the schedule when considering an upgrade; you may want to wait until *after* the next release.

The Element of Surprise

When you find yourself surprised by a bug (perhaps even muttering "that's impossible" under your breath where we can't hear you), you must reevaluate truths you hold dear. In that discount calculation algorithm—the one you knew was bulletproof and couldn't possibly be the cause of this bug—did you test *all* the boundary conditions? That other piece of code you've been using for years—it couldn't possibly still have a bug in it. Could it?

Of course it can. The amount of surprise you feel when something goes wrong is proportional to the amount of trust and faith you have in the code being run. That's why, when faced with a "surprising" failure, you must accept that one or more of your assumptions is wrong. Don't gloss over a routine or piece of code involved in the bug because you "know" it works. Prove it. Prove it in *this* context, with *this* data, with *these* boundary conditions.

Tip 34 Don't Assume It—Prove It

When you come across a surprise bug, beyond merely fixing it, you need to determine why this failure wasn't caught earlier. Consider whether you need to amend the unit or other tests so that they would have caught it.

Also, if the bug is the result of bad data that was propagated through a couple of levels before causing the explosion, see if better parameter checking in those routines would have isolated it earlier (see the discussions on crashing early and assertions on page 113 and on page 115, respectively).

While you're at it, are there any other places in the code that may be susceptible to this same bug? Now is the time to find and fix them. Make sure that *whatever* happened, you'll know if it happens again.

If it took a long time to fix this bug, ask yourself why. Is there anything you can do to make fixing this bug easier the next time around? Perhaps you could build in better testing hooks, or write a log file analyzer.

Finally, if the bug is the result of someone's wrong assumption, discuss the problem with the whole team: if one person misunderstands, then it's possible many people do.

Do all this, and hopefully you won't be surprised next time.

Debugging Checklist

- Is the problem being reported a direct result of the underlying bug, or merely a symptom?
- Is the bug *really* in the framework you're using? Is it in the OS? Or is it in your code?
- If you explained this problem in detail to a coworker, what would you say?
- If the suspect code passes its unit tests, are the tests complete enough? What happens if you run the tests with *this* data?
- Do the conditions that caused this bug exist anywhere else in the system? Are there other bugs still in the larval stage, just waiting to hatch?

Related Sections Include

- Topic 24, *Dead Programs Tell No Lies*, on page 112

Challenges

- Debugging is challenge enough.

21

Text Manipulation

Pragmatic Programmers manipulate text the same way woodworkers shape wood. In previous sections we discussed some specific tools—shells, editors, debuggers—that we use. These are similar to a woodworker's chisels, saws, and planes—tools specialized to do one or two jobs well. However, every now and then we need to perform some transformation not readily handled by the basic tool set. We need a general-purpose text manipulation tool.

Text manipulation languages are to programming what routers[5] are to woodworking. They are noisy, messy, and somewhat brute force. Make mistakes

5. Here *router* means the tool that spins cutting blades very, very fast, not a device for interconnecting networks.