chapter

1

Introduction

''At present I am, as you know, fairly busy, but I propose to devote my declining years to the composition of a textbook which shall focus the whole art of detection into one volume.''

—SHERLOCK HOLMES, *THE ADVENTURE OF THE ABBEY GRANGE*

This book tells you how to find out what's wrong with stuff, quick. It's short and fun because it *has* to be—if you're an engineer, you're too busy debugging to read anything more than the daily comics. Even if you're not an engineer, you often come across something that's broken, and you have to figure out how to fix it.

Now, maybe some of you never need to debug. Maybe you sold your dot.com IPO stock before the company went belly-up and you simply have your people look into the problem. Maybe you always luck out and your design just works—or, even less likely, the bug is always easy to find. But the odds are that you and all your competitors have a few hard-to-find bugs in your designs, and whoever fixes them quickest has an advantage. When you can find bugs fast, not only do you get quality products to customers quicker, you get yourself home earlier for quality time with your loved ones.

So put this book on your nightstand or in the bathroom, and in two weeks you'll be a debugging star.

How Can That Work?

How can something that's so short and easy to read be so useful? Well, in my twenty-six years of experience designing and debugging systems, I've discovered two things (more than two, if you count stuff like ''the first cup of coffee into the pot contains all the caffeine''):

1. When it took us a long time to find a bug, it was because we had neglected some essential, fundamental rule; once we applied the rule, we quickly found the problem.
2. People who excelled at quick debugging inherently understood and applied these rules. Those who struggled to understand or use these rules struggled to find bugs.

I compiled a list of these essential rules; I've taught them to other engineers and watched their debugging skill and speed increase. They really, really work.

Isn't It Obvious?

As you read these rules, you may say to yourself, ''But this is all so obvious.'' Don't be too hasty; these things *are* obvious (fundamentals usually are), but how they apply to a particular problem isn't always so obvious. And don't confuse obvious with easy—these rules aren't always easy to follow, and thus they're often neglected in the heat of battle.

The key is to *remember* them and *apply* them. If that was obvious and easy, I wouldn't have to keep reminding engineers to use them,

and I wouldn't have a few dozen war stories about what happened when we didn't. Debuggers who naturally use these rules are hard to find. I like to ask job applicants, ''What rules of thumb do you use when debugging?'' It's amazing how many say, ''It's an art.'' Great—we're going to have Picasso debugging our image-processing algorithm. The easy way and the artistic way do not find problems quickly.

This book takes these ''obvious'' principles and helps you remember them, understand their benefits, and know how to apply them, so you can resist the temptation to take a ''shortcut'' into what turns out to be a rat hole. It turns the art of debugging into a science.

Even if you're a very good debugger already, these rules will help you become even better. When an early draft of this book was reviewed by skilled debuggers, they had several comments in common: Besides teaching them one or two rules that they weren't already using (but would in the future), the book helped them crystallize the rules they already unconsciously followed. The team leaders (good debuggers rise to the top, of course) said that the book gave them the right words to transmit their skills to other members of the team.

Anyone Can Use It

Throughout the book I use the term *engineer* to describe the reader, but the rules can be useful to a lot of you who may not consider yourselves engineers. Certainly, this includes you if you're involved in figuring out what's wrong with a design, whether your title is engineer, programmer, technician, customer support representative, or consultant.

If you're not directly involved in debugging, but you have responsibility for people who are, you can transmit the rules to your people. You don't even have to understand the details of the systems and tools your people use—the rules are fundamental, so after read-

ing this book, even a pointy-haired manager should be able to help his far-more-intelligent teams find problems faster.

If you're a teacher, your students will enjoy the war stories, which will give them a taste of the real world. And when they burst onto that real world, they'll have a leg up on many of their more experienced (but untrained in debugging) competitors.

It'll Debug Anything

This book is general; it's not about specific problems, specific tools, specific programming languages, or specific machines. Rather, it's about universal techniques that will help you to figure out any problem on any machine in any language using whatever tools you have. It's a whole new level of approach to the problem—for example, rather than tell you how to set the trigger on a Glitch-O-Matic digital logic analyzer, I'm going to tell you *why* you have to use an analyzer, even though it's a lot of trouble to hook it up.

It's also applicable to fixing all kinds of problems. Your system may have been designed wrong, built wrong, used wrong, or just plain got broken; in any case, these techniques will help you get to the heart of the problem quickly.

The methods presented here aren't even limited to engineering, although they were honed in the engineering environment. They'll help you figure out what's wrong with other things, like cars, houses, stereo equipment, plumbing, and human bodies. (There are examples in the book.) Admittedly, there are systems that resist these techniques—the economy is too complex, for example. And some systems don't need these methods; e.g., everybody already *knows* what's wrong with the government.

But It Won't Prevent, Certify, or Triage Anything

While this book is general about methods and systems, it's very focused on *finding the causes of bugs and fixing them.*

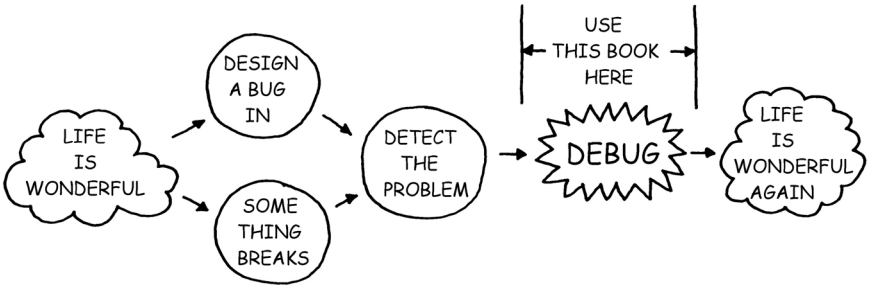
It's not about quality development processes aimed at preventing bugs in the first place, such as ISO-9000, code reviews, or risk management. If you want to read about that, I recommend books like *The Tempura Method of Totalitarian Quality Management Processes* or *The Feng Shui Guide to Vermin-Free Homes*. Quality process techniques are valuable, but they're often not implemented; even when they are, they leave some bugs in the system.

Once you have bugs, you have to detect them; this takes place in your quality assurance (QA) department or, if you don't have one of those, at your customer site. This book doesn't deal with this stage either—test coverage analysis, test automation, and other QA techniques are well handled by other resources. A good book of poetry, such as *How Do I Test Thee, Let Me Count the Ways*, can help you while away the time as you check the 6,467,826 combinations of options in your product line.

And sooner or later, at least one of those combinations will fail, and some QA guy or customer is going to write up a bug report. Next, some managers, engineers, salespeople, and customer support people will probably get together in a triage meeting and argue passionately about how important the bug is, and therefore when and whether to fix it. This subject is deeply specific to your market, product, and resources, and this book will not touch it with a ten-foot pole. But when these people decide it has to be fixed, you'll have to look at the bug report and ask yourself, ''How the heck did that happen?'' That's when you use this book (see Figure 1-1).

The following chapters will teach you how to prepare to find a bug, dig up and sift through the clues to its cause, home in on the

Figure 1-1. When to Use This Book.



actual problem so you can fix it, and then make sure you really fixed it so you can go home triumphant.

More Than Just Troubleshooting

Though the terms are often interchanged, there's a difference between debugging and troubleshooting, and there's a difference between this debugging book and the hundreds of troubleshooting guides available today. Debugging usually means figuring out why a design doesn't work as planned. Troubleshooting usually means figuring out what's broken in a particular copy of a product when the product's design is known to be good—there's a deleted file, a broken wire, or a bad part. Software engineers debug; car mechanics troubleshoot. Car *designers* debug (in an ideal world). Doctors troubleshoot the human body—they never got a chance to debug it. (It took God one day to design, prototype, and release that product; talk about schedule pressure! I guess we can forgive priority-two bugs like bunions and male pattern baldness.)

The techniques in this book apply to both debugging and troubleshooting. These techniques don't care how the problem got in there;

they just tell you how to find it. So they work whether the problem is a broken design or a broken part. Troubleshooting books, on the other hand, work *only* on a broken part. They boast dozens of tables, with symptoms, problems, and fixes for anything that might go wrong with a particular system. These *are* useful; they're a compendium of everything that has ever broken in that type of system, and what the symptoms and fixes were. They give a troubleshooter the experience of many others, and they help in finding known problems faster. But they don't help much with new, unknown problems. And thus they can't help with design problems, because engineers are so creative, they like to make up new bugs, not use the same old ones.

So if you're troubleshooting a standard system, don't ignore Rule 8 (''Get a Fresh View''); go ahead and consult a troubleshooting guide to see if your problem is listed. But if it isn't, or if the fix doesn't work, or if there's no troubleshooting guide out yet because you're debugging the world's first digital flavor transmission system, you won't have to worry, because the rules in this book will get you to the heart of your brand-new problem.

A Word About War Stories

I'm a male American electronics engineer, born in 1954. When I tell a ''war story'' about some problem that got solved somehow, it's a real story, so it comes from things that male American electronics engineers born in 1954 know about. You may not be all or any of those, so you may not understand some of the things I mention. If you're an auto mechanic, you may not know what an interrupt is. If you were born in 1985, you may not know what a record player is. No matter; the principle being demonstrated is still worth knowing, and I'll explain enough as I go along so you'll be able to get the principle.

You should also know that I've taken some license with the details to protect the innocent, and especially the guilty.

Stay Tuned

In this book I'll introduce the nine golden rules of debugging, then devote a chapter to each. I'll start each chapter with a war story where the rule proved crucial to success; then I'll describe the rule and show how it applies to the story. I'll discuss various ways of thinking about and using the rule that are easy to remember in the face of complex technological problems (or even simple ones). And I'll give you some variations showing how the rule applies to other stuff like cars and houses.

In the final few chapters, I've included a set of war stories to exercise your understanding, a section on using the rules under the trying circumstances of the help desk, and a few last hints for putting what you've learned to work in your job.

When you're done with this book, your debugging efficiency will be much higher than before. You may even find yourself wandering around, looking for engineers in distress so you can swoop in and save the day. One bit of advice, though: Leave the leotard and cape at home.