Extracted from:

Debug It!

Find, Repair, and Prevent Bugs in Your Code

The Pragmatic Programmers

Debug It!

Find, Repair, & Prevent Bugs in Your Code

Paul Butcher

Edited by Jacquelyn Carter

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-28-X

ISBN-13: 978-1-934356-28-9

Printed on acid-free paper.

B1.0 printing, June 17, 2009

Version: 2009-6-16

Chapter 3

Diagnose

Diagnosis is *the* key element of debugging. This is where the rubber meets the road and you arrive at the understanding of the root cause of the behavior you're seeing.

In this chapter, we will cover:

- The core diagnostic process.
- Different types of experiment, and what makes a good experiment.
- Useful stratagems.

3.1 Stand Back—I'm Going to Try Science

Although you're going to be using various tools and techniques, and leveraging your software itself to help you, your primary asset is and always will be your intellect. Diagnosis takes place within your mind, not within your computer.

Balance creativity with rigor

The mindset one needs to cultivate when debugging is similar (because the problem is similar) to that of a detective solving a crime or a scientist investigating a new phenomenon. Open-minded at the same time as methodical, creative at the same time as thorough—as with so many other aspects of software development, effective bug fixing is all about finding the appropriate balance between these apparently contradictory demands.

The scientific method can work in two different directions.[1] In one case, we start with a hypothesis and attempt to create experiments, the results of which will either support or refute it. In the other, we start with an observation that doesn't fit with our current theory and as a result modify that theory or possibly even replace it with something completely different.

In debugging, we almost always start from the latter. Our theory (that the software behaves as we think it does) is disproved by an observation (the bug) that demonstrates that we are mistaken. In the words of Thomas Huxley, "The great tragedy of Science—the slaying of a beautiful hypothesis by an ugly fact."

A Debugging Method

Having discovered that things aren't as you believed them to be, your task is to modify your understanding of the software until you *do* understand what's really going on. To do that, you operate in the other of the two possible directions—create a hypothesis that might provide an explanation and then construct experiments to test it.

So here's our idealized process (see Figure 3.1, on the next page):

1. Examine what you know about the software's behavior and construct a hypothesis about what might cause it.
2. Design an experiment that will allow you to test its truth (or otherwise).
3. If the experiment disproves your hypothesis, come up with a new one and start again.
4. If it supports your hypothesis, keep coming up with experiments until you have either disproved it, or reached a high enough level of certainty to consider it proven.

All well and good, but rather abstract. How do you translate this into action?

Different Types of Experiment

Your starting point is the reproduction we discussed at length in the last chapter. From that starting point, there are several different types

1. Students of the History and Philosophy of Science will realize that I am skating over many subtleties.

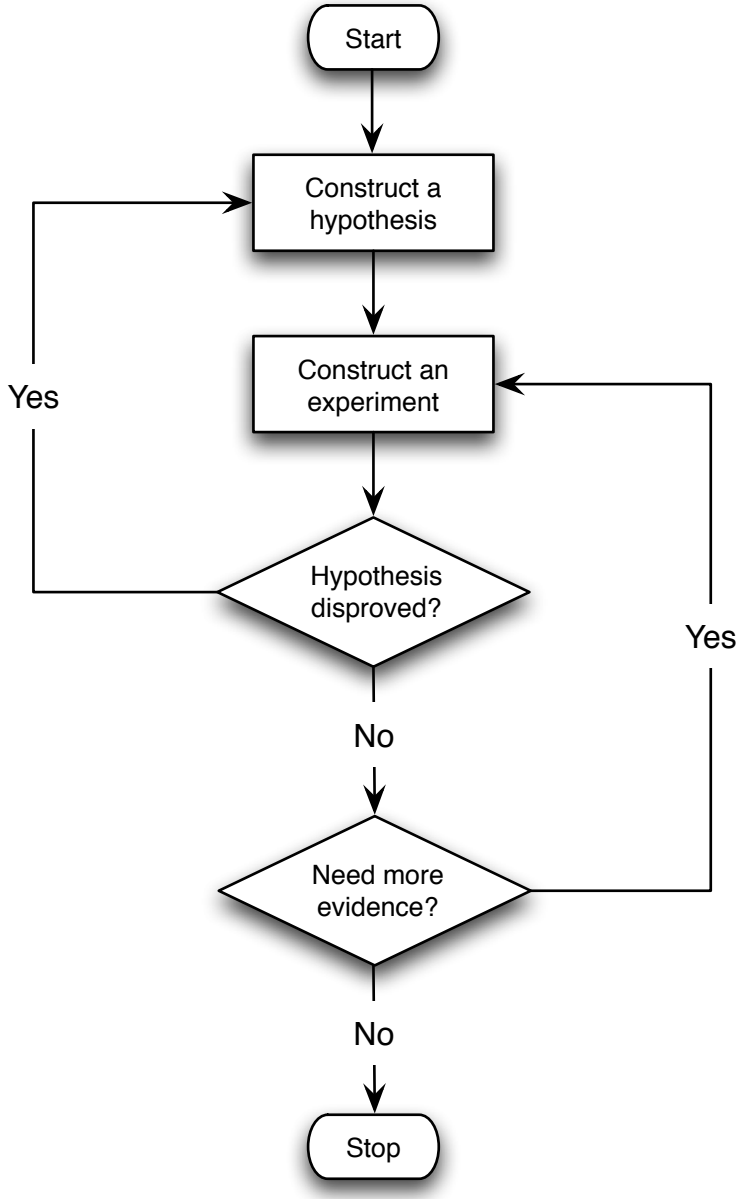

Figure 3.1: A Debugging Method

of experiment that you can run—each of which involves changing one aspect of how you reproduce the problem:

- You can examine some aspect of the software's internal state (either by instrumenting it directly or by running it under a debugger).
- You can modify some aspect of how you run the software (modified inputs, for example, or an alternative environment) and see if it behaves differently.
- You can change the logic encoded within the software itself and examine the effect of that change.

Which of these you choose depends upon the nature of your hypothesis, and making the best choice comes down to experience and intuition.

Whichever you choose, however, the most important thing to bear in mind is that your experiment must have a clear goal.

Experiments Must Prove Something

Experiments are a means to an end, not an end in themselves. There is no point performing an experiment unless it proves something.

What is your experiment going to tell you?

Before investing time and effort to construct and run an experiment, ask yourself what it's going to tell you. What are the possible outcomes? If none of those outcomes would move you closer to your diagnosis, you need to come up with a different experiment. Beware of confusing activity with progress—if an experiment cannot increase your understanding, it's a waste of your time.

You can design experiments that are intended to *prove* your hypothesis, or to *disprove* it. It might seem counter-intuitive, but frequently the latter are the more useful. In part, this is because it's difficult to incontrovertibly prove something (just because you see what you expect to see doesn't mean that you're seeing it for the reason you think you are), but mainly it's a question of psychology.

If you have a plausible explanation for what's happening, it's very easy to talk yourself into seeing what you want to see. Playing devil's advocate and trying to disprove your hypothesis can be very productive, helping you spot possible holes in the explanation that you wouldn't see otherwise. If, after you've tried your hardest to disprove it, its still standing at the end, then you can have a lot of confidence that you've

nailed it. And every once in a while you will surprise yourself and find that something very different from what you thought is happening.

One Change at a Time

One of the basic rules of constructing experiments is that you should only make a single change at a time.

Multiple changes lead to misleading conclusions

If you make a single change and see an effect, you can be pretty certain that the one caused the other.[2] If you make more than one change, however, it can be very difficult to be sure which change resulted in which effect. Or the changes may interact in unpredictable ways. At best, this might mean that you are unable to conclude anything useful. At worst you may reach misleading conclusions that lead you down completely the wrong path.

This rule applies to any kind of change—changes to the source, the environment, input files and so on. Anything, in fact, that might have an effect on the software.

For some reason this principle is forgotten surprisingly frequently—I don't know how many times I've seen someone make several changes all at once and then try to make sense of the results afterwards. Although it can feel as though you're saving yourself time by making several changes simultaneously, all that you really achieve is the risk of invalidating your results. Maintain your discipline and avoid falling into this trap.

Finally, once you see a change in behavior, undo whatever apparently caused it and verify that the behavior returns to what it was beforehand. This is a very powerful indication that you're looking at cause and effect rather than serendipity.

Keep a Record of What You've Tried

If you find yourself working on a bug that takes days or weeks to track down, you will end up carrying out many different experiments. Ideally each one will eliminate a set of possible causes and eventually you will zero in on the root cause.

2. Not completely certain—a changing underlying system can get in the way of this kind of reasoning—but it's an excellent starting hypothesis.

When the diagnosis goes on this long and involves this many experiments, there is a danger that you will lose track of what you've done. This may mean that you waste time investigating possibilities that have already been eliminated by previous experiments, or it could result in you heading down a blind alley. In the worst case, it could lead you to a broken conclusion and subsequent misdiagnosis.

Periodically review what you've already tried and learned

The best defense is to maintain a record of the experiments you've tried, and what the results were. This doesn't have to take a long time or include huge amounts of detail—just enough to ensure that you don't forget what you've already done. Periodically review your notes to refresh your memory and help you identify the most promising next steps.

Many developers find it helpful to maintain a *daybook*. They might use it to record notes from meetings, design sketches, a record of the steps necessary to install a piece of software—anything, in fact, that might prove useful to refer to in the future. A daybook can be an excellent place to record your experiments. Or alternatively, if you prefer to keep your notes electronically, you might consider keeping a personal Wiki.

Ignore Nothing

Occasionally you will notice odd behavior. You run an experiment, expecting one of result A or result B, and instead get result C. Or you work through a set of instructions about how to reproduce the bug, and the software does something very different from what you expect.

It can sometimes be tempting to shrug it off as "one of those things" and try a different tack. Don't! The software is trying to tell you something and it's in your interest to listen.

If something unexpected happens, it means that some assumption you're making is broken. This might be an assumption about how the software should behave, what the bug you're trying to hunt down is, how you've constructed your experiment, or anything else. If you have a broken assumption, then the most valuable thing that you can do is to stop, identify and fix it. If you don't, then all bets are off and you can't trust any conclusions you reach.

This kind of thing can turn out to be a blessing in disguise—a short-cut to what's really going on. Getting to the bottom of unexpected behavior can save you a huge amount of wasted time chasing will-o-the-wisps.

Joe Asks...

How else Can a Daybook Help when Debugging?

As well as maintaining a record of your experiments, a daybook can also be useful for:

- Writing out hypotheses. Getting things onto paper can help identify flaws in assumptions, especially when the hypothesis is complex.
- Keeping track of details like stack traces, argument values and variable names. Not only does this help with finding things again, but it also helps you communicate with colleagues when explaining the problem, avoiding the need to rely upon memory.
- Keeping a list of ideas to try. Often you will notice something else you want to investigate, or a possible followup experiment will occur to you, but you don't want to abandon the current experiment to pursue it. A "to do" list ensures that you don't forget to come back to it later.
- Doodling when you need to take your mind off the problem.

Anything that you don't understand is potentially a bug

Even if the odd behavior you notice doesn't have any bearing on the problem at hand, the fact that you've discovered something unexpected is valuable. Anything that you don't understand is potentially a bug. Once you've demonstrated to your satisfaction that it isn't relevant to what you're working on, feel free to put it aside, but don't forget about it. Keep a record (file a bug report, perhaps) and come back to it. Very often, things discovered in passing like this prove to be real issues that need fixing. And you would much rather fix them having discovered them this way, than wait until they're reported by an irate customer.

Sneaky!

I was crawling through yesterday's server logfile gathering evidence that would help me diagnose the problem I was working on. In passing, I noticed that one of our users seemed to be having connection problems—he was logging out and then back in over and over again.

This had nothing whatsoever to do with the problem I was chasing, and I very nearly let it pass. Connection problems aren't that unusual, after all. But something didn't feel right—the pattern was too regular. My "spidey sense" was tingling.

Sure enough, it turns out that the user in question had found a sneaky way to bypass one of the security mechanisms implemented by the software (which rationed how much of a certain resource each user could consume). By logging out and then immediately back in again, he could reset his quota. An easy bug to fix, now that we knew about it.

3.2 Stratagems

Although every bug is different, there are certain techniques and approaches that have repeatedly proven their value in tracking down a wide range of problems. They won't suffice for every problem you find yourself faced with, but every programmer should have them at their fingertips.

Instrumentation

Diagnosis is all about information—divining precisely the state of, and the execution path taken by the software. Although there are many ways through which you can either infer or derive this information, by far the simplest and most direct is adding *instrumentation* to the software itself.

Instrumentation is code that doesn't affect how the software behaves, but instead provides insight into why it behaves as it does. In the last chapter, we already discussed the most common and important type of instrumentation, logging. Possibly the oldest debugging technique is adding ad-hoc logging to the code,[3] in order to confirm or refute our beliefs about what it's doing.

The full facilities of the language are at your disposal

Instrumentation isn't limited to simple output statements, however—you have the full facilities of the language at your disposal. You can collect and collate data, evaluate arbitrary code and test for relevant conditions—the only limit is your imagination.

3. Often called printf() debugging after the C function of the same name.

Beware of Heisenberg

One of the lessons of quantum physics is that the act of observing a system can change the system itself. Computer software isn't quantum mechanical (not yet, anyway) but we still need to be wary.

Instrumenting software intrinsically involves changing it, which raises the specter of affecting, instead of simply observing, its behavior. This is dangerous during diagnosis, because introducing an unintentional change during a series of experiments can easily lead to you draw invalid conclusions.

Fundamentally speaking, there is no way that you can guarantee to avoid introducing *some* side-effects. The fact that you've modified the source code means that the layout of the object code in memory and the timing of its execution will be affected. Happily, most of the time this remains a purely hypothetical problem—as long as you're careful to avoid the more obvious side-effects, you can normally ignore the issue.

Nevertheless, it is very good practice to keep the source code as close to its pristine form as possible. Don't allow failed experiments, along with their possible side-effects, to accumulate over time. Keeping things neat also helps ensure that the code remains easy (or at least, no harder) to understand and will help to ensure that you don't check in unintended changes when you eventually come to fixing the problem.

Let's look at an example. Imagine that you're trying to track down a bug in some Java code that traverses a data structure, processing each node in turn:

```
while(node != null) {
  node.process();
  node = node.getNext();
}
```

You're seeing behavior that suggests that nodes are being processed more than once (in other words, getNext() is returning one or more nodes more than once). It's not clear which nodes are being processed more than once, however. One way to find the problem would be to instrument the code as follows:

```
❶ HashSet processed = new HashSet();
```

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Debug It!'s Home Page
http://pragprog.com/titles/pbdp
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/pbdp.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)