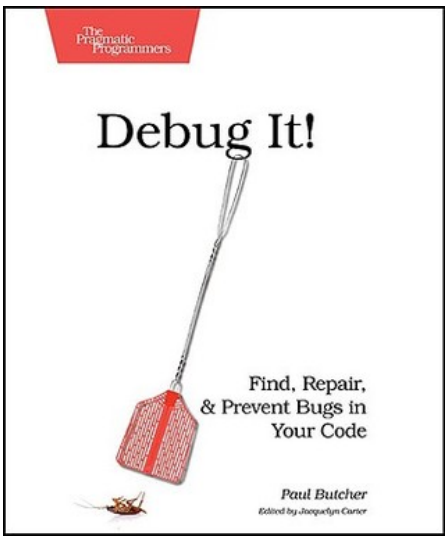

Debug It!: Find, Repair, and Prevent Bugs in Your Code

Paul Butcher

Debug It!: Find, Repair, and Prevent Bugs in Your Code

Paul Butcher

Debug It!: Find, Repair, and Prevent Bugs in Your Code Paul Butcher
Some developers thrash around aimlessly looking for a bug without concrete results. Others have the knack of unerringly zeroing in on the root cause of a bug. Are they geniuses? Just lucky? No, they've learned the secrets of professional debugging. This book will equip you with the tools, techniques and approaches-proven in the crucible of professional software development-to ensure that you can tackle any bug with confidence.

You'll learn how to handle every stage of the bug life-cycle, from constructing software that makes debugging easy, through detection, reproduction, diagnosis and rolling out your eventual fix.

If you develop software, sooner or later you're going to discover that it doesn't always behave as you intended. Working out why it's misbehaving can be hard. Sometimes **very** hard. *Debug It!* is here to help!

All bugs are different: there is no silver bullet. You've got to rely upon your intellect, intuition, detective skills and yes, even a little luck. But that doesn't mean that you're completely on your own-there is much you can learn from those who have gone before. This book distills decades of hard-won experience gained in the trenches of professional software development, giving you a head-start and arming you with the tools you need to get to the bottom of the problem, whatever you're faced with.

Whether you're writing Java or assembly language, targeting servers or embedded micro-controllers, using agile or traditional approaches, the same basic bug-fixing principles apply. From constructing software that is easy to debug (and incidentally less likely to contain bugs in the first place), through handling bug reports to rolling out your ultimate fix, we'll cover the entire life-cycle of a bug.

You'll learn about the empirical approach, which leverages your software's unique ability to **show** you what's really happening, the importance of finding a reliable and convenient means of reproducing a bug, and common pitfalls so you can avoid them. You'll see how to use commonly available tools to automatically detect problems before they're reported by customers and how to construct "transparent software" that provides access to critical information and internal state.

Debug It!: Find, Repair, and Prevent Bugs in Your Code Details

Date	:	Published November 22nd 2009 by Pragmatic Bookshelf (first published 2009)
ISBN	:	9781934356289
Author	:	Paul Butcher
Format	:	Paperback 232 pages
Genre	:	Computer Science, Programming, Technical, Computers, Software, Science, Technology

From Reader Review Debug It!: Find, Repair, and Prevent Bugs in Your Code for online ebook

Rod Hilton says

I loved the idea of this book. So many aspects of software development have books written about them. How to write good tests, how to design good objects, how to increase performance, etc. But there don't seem to be any books written about the art of debugging. I was excited to grab a copy of Debug It! and fine-tune my debugging skills.

I was tremendously disappointed. If you are skeptical of the book, thinking "but I've been debugging for a while, what more is there to learn about it," let me assure you that your concerns are right on. The book contains almost no practical advice for actually debugging. It contains references to a few types of bugs you might see, suggests setting up experiments, and advises only changing one thing at a time. Any programmer who has been in the business for more than a year knows all of this.

The author seems to simply have nothing to say about the topic, to be honest. A particular section on bugs that are particularly tricky contained no useful information, and actually suggested to the reader to "keep at it, you'll figure it out eventually." Seriously? Is this a technical book or a self-help audiobook?

The author has two or three blog posts worth of material, which he has stretched out to an entire 200-page book via useless anecdotes and insipid, generic advice. I hate to sound so harsh, but this book simply has no reason to exist. The second half is slightly better than the first, so I'm glad I forced my way through it, but I found myself skimming large portions after a while, able to tell from as little as the section title that I could skip it.

There's an entire chapter about anti-patterns, which I did enjoy, but none of the antipatterns had the slightest thing to do with debugging. It clearly belonged in a different (better) book. The parts of the book related to debugging are superficial and light on real content. I cannot recommend this book for anyone other than a recent college graduate who has yet to actually debug anything at a real job.

Sundarraj Kaushik says

Debug It! by Paul Butcher

In the book the author covers different aspects of

Effective Debugging

In the first chapter the author advises
1. Work out why the software is behaving unexpectedly.
2. Fix the problem.
3. Avoid breaking anything else.
4. Maintain or improve the overall quality (readability, architecture, test coverage, performance, and so on) of the code.

5. Ensure that the same problem does not occur elsewhere and cannot occur again.

The author emphasizes that without first understanding the true root cause of the bug, we are outside the realms of software engineering and delving instead into voodoo programming or programming by coincidence.

He suggests that empirical means is the best way to Debug i.e. provide different inputs and observe how the system behaves.

The Core Debugging Process involves the following steps:
1. Reproduce: Find a way to reliably and conveniently reproduce the problem on demand.
2. Diagnose: Construct hypotheses, and test them by performing experiments until you are confident that you have identified the underlying cause of the bug.
3. Fix: Design and implement changes that fix the problem, avoid introducing regressions, and maintain or improve the overall quality of the software.
4. Reflect: Learn the lessons of the bug. Where did things go wrong? Are there any other examples of the same problem that will also need fixing? What can you do to ensure that the same problem doesn't happen again?

Address one Bug at a time: Picking too many bugs to address at one time will prevent focus on one.
Check Simple Things first: Somebody may have encountered something similar and may already have a solution.

Reproduce
1. Reproduction of the error should consistent and efficient, otherwise testing the fixing will become a botheration.
2. So reproduce the error in an controlled the environment to achieve consistency.
3. To keep it efficient try and reduce the input to be provided and reduce the processing that needs to be done, store the state at every step so that only the errorneous step needs to be rerun.
4. Automate the test conditions to make it quicker and easier to test the application after the fix. Replaying the log file can be a good strategy in scenarios where logging using proxy was used to capture the error condition.

The following will help in reproducing the error:
1. Logging at appropriate places so that one knows what is happening in the system. Too much logging will be unacceptable in a production system.
2. Where possible usage of a proxy to capture the network traffic and try to reproduce the error with this traffic.
3. If calls to libraries are problematic or they need to be emulated in a test environment, write a Shim (a proxy to a library) and capture the inputs and outputs and use this to reproduce the error. In engineering, a shim is a thin piece of material used to fill the space between objects. In computing we've borrowed the term to mean a small library that sits between a larger library and its client code. It can be used to convert one API to another or, as in the case we're discussing here, to add a small amount of functionality without having to modify the main library itself.
4. Reach out the user community that is able to reproduce the error and get inputs from them. Give them specially instrumented code to figure out the error.
5. Read the documentation on the system, if the problem seems to be occurring beyond the realms of the code that has been written, and read the errors reported by others using the same platform.

Irreproducible Errors

Most bugs are reproducible. The few scenarios where the bug may be irreproducible or difficult to reproduce will be because of the following reasons:

1. Starting from an unpredictable initial state: C, C++ programs are prone to this error.
2. Interaction with external systems: This can happen if the other system is no running in lock-step with this software. If inputs from this external systems arrives when the current system is under different states the error can be difficult to reproduce.
3. Deliberate Randomness: In some systems there is deliberate randomness as in games. These can be difficult to debug. But if the same seed is used for the pseudo-random number generator then the bug will become easier to reproduce.
4. Multithreading - This happens because of the pre-emptive multi-tasking provide by the Operating System. Since the threads can be stalled and restarted at different times depending on the activity in the CPUs at that that time, it becomes difficult to reproduce errors in such an environment. Trying using the sleep to try and simulate the stalling of one thread and execution of another to try and emulate the error.

Good Practices of reproducing

If a bug takes a long time and is still not identified this may be because another bug is masking this one. So try to concentrate on a different bug in the same area and possibly clear it before retrying the difficult one.

Diagnosis

How to Diagnose?

1. Examine what you know about the software's behaviour, 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.

Techniques of Diagnosing

1. Instrument the code to understand the flow better.
2. Use a binary search pattern and logging to locate the source code of error. I.e. look for error before and after the execution of a stretch of code. If error is found now look for the error in the first half of this code stretch, if not found then look for the error in the second half of the stretch; then further split the stretch found into further two halves; repeat this until the exact point of error is found.
3. Use a binary search pattern in version control to identify the version when error was introduced.
4. Use a binary search pattern on data to identify the version of the error.
5. Focus on the differences. The Application works for most customers, but not for specific ones. Check how these customers are different from the rest where the application is working. Similarly works in most environments, but does not work in a particular environment. Try and figure out what is different in that environment. If it happens for specific input files then figure out what is different in that file as compared to other files where it works.
6. Use debuggers when available.
7. Use the Interactive Consoles where debuggers are not available or are not good.

Good Practices of Diagnosing

1. When experimenting make only one change at a time.

2. Ignore nothing. Do not shrug off the unexpected as and anomaly. It could be that our assumptions are wrong.
3. Maintain a record of experiments and results so that it is easy to trace back.
4. Anything that you don’t understand is potentially a bug.
5. Learn from others. Search in the net for similar problem and solution offered.
6. All other things being equal, the simplest explanation is the best. - Occam's Razor
7. Writing Automated Test Cases helps because this lets us concentrate only on broker cases.
8. Keep Asking "Are you changing the right thing?" If the changes you’re making have no effect, you’re not changing what you think you are.
9. Validate and revalidate your assumptions
10. Ensure that the underlying system on which diagnosis is being done is static and not changing.
11. If one is stuck in debugging a problem, one good way is to ask somebody else to take a look at it.

Fixing

Best Practices during fixing
1. Make sure you know how you’re going to test it before designing your fix.
2. Do not let the fixes mess up with the original clean design and structure of code. Haphazardly put together fixes can mess up the good design principles followed in the original design. Any fix should leave the code in better shape than it was before.
3. Clean up any adhoc code changed before making the final fix so that no unwanted code gets checked in. Keep only what is absolutely necessary.
4. Use existing test cases. Modify the test cases if required or write the failing test case and test code without the fix. Then fix the code and test the failing test case to see that it passes after the fix.
1. Run the existing tests, and demonstrate that they pass.
2. Add one or more new tests, or fix the existing tests, to demonstrate the bug (in other words, to fail).
3. Fix the bug.
4. Demonstrate that your fix works (the failing tests no longer fail).
5. Demonstrate that you haven’t introduced any regressions (none of the tests that previously passed now fail).
5. Fix the Root Cause not the symptom. E.g. if one encounters a NullPointerException, the solution is not to capture the NullPointerException and handle or even worse suppress it, it is necessary to figure out why the NullPointerException is occurring and fixing that cause. Giving into temptation of quick fixes is not the right thing, making the right fix is the right thing.
6. Refactor or change functionality or fix a problem — one or the other, never more than one.
7. Always check in small changes. Do not check in large changes as it will make it very difficult to find out which change actually caused the problem. Ensure check-in comments are as meaningful (and specific) as possible.
8. Diff and check what exactly is being checked in before actually checking in.
9. Get the code reviewed. This is very important as unnoticed errors

After Fixing - Reflect

Sometimes "The six stages of debugging" reads as follows:
1. That can’t happen.
2. That doesn’t happen on my machine.
3. That shouldn’t happen.
4. Why is that happening?
5. Oh, I see.

6. How did that ever work?

After fixing one needs to reflect on the following points:
• How did it ever work?
• When and why did the problem slip through the cracks?
• How to ensure that the problem never happens again?

Find out the root cause. A useful trick when performing root cause analysis is to ask
“Why?” five times. For example:
• The software crashed. Why?
• The code didn’t handle network failure during data transmission. Why?
• There was no unit test to check for network failure. Why?
• The original developer wasn’t aware that he should create such a test. Why?
• None of our unit tests check for network failure. Why?
• We failed to take network failure into account in the original design.

After fixing do the following:
1. Take steps to ensure that it does not ever happen again. Educate yourself, educate others on the team.
2. Check if there are other similar errors.
3. Check if the documentation needs to be updated as a result of the fix.

Other aspects of handling and managing bugs
1. To better aid debugging collect relevant environment and configuration information automatically.
2. Detect bugs early, and do so from day one.
3. Poor quality is contagious. Broken Window concept. The theory was introduced in a 1982 article by social scientists James Q. Wilson and George L. Kelling. So do not leave bad code. Fix bad code at the earliest.
4. A Zero Bug Software is impossible, so take a pragmatic approach and try to reach as close to Zero bugs as possible. Temper perfectionism with pragmatism.
5. Keep the design simple. Not only does a simple design make your software easier to understand and less likely to contain bugs in the first place, it also makes it easier to control—which is particularly useful when trying to reproduce problems in concurrent software.
6. Automate your entire build process, from start to finish.
7. Version management of code is absolutely mandatory.
7. Different source should mean different version number. Even if the change to the code is minuscule.

These are excerpts from the book. These are the summary from end of each of the chapter.
Chapter 1 - A Method in Madness
• Make sure to do the following:
– Work out why the software is behaving unexpectedly.
– Fix the problem.
– Avoid breaking anything else.
– Maintain or improve overall quality.
– Ensure that the same problem does not occur elsewhere and cannot occur again.
• Leverage your software’s ability to show you what’s happening.
• Work on only one problem at a time.
• Make sure that you know exactly what you’re looking for:
– What is happening?
– What should be happening?
• Check simple things first.

Chapter 2 - Reproduce
• Find a reproduction before doing anything else.
• Ensure that you're running the same version as the bug was reported against.
• Duplicate the environment that the bug was reported in.
• Determine the input necessary to reproduce the bug by:
– Inference
– Recording appropriate inputs via logging
• Ensure that your reproduction is both reliable and convenient
through iterative refinement:
– Reduce the number of steps, amount of data, or time required.
– Remove nondeterminism
- Automate.

Chapter 3 - Diagnose
• Construct hypotheses, and test them with experiments.
– Make sure you understand what your experiments are going to tell you.
– Make only one change at a time.
– Keep a record of what you've tried.
– Ignore nothing.
• When things aren't going well:
– If the changes you're making don't seem to be having an effect, you're not changing what you think you are.
– Validate your assumptions.
– Are you facing multiple interacting causes or a changing underlying system?
• Validate your diagnosis.

Chapter 4 - Fix
• Bug fixing involves three goals:
– Fix the problem.
– Avoid introducing regressions.
– Maintain or improve overall quality (readability, architecture, test coverage, and so on) of the code.
• Start from a clean source tree.
• Ensure that the tests pass before making any changes.
• Work out how you're going to test your fix before making changes.
• Fix the cause, not the symptoms.
• Refactor, but never at the same time as modifying functionality.
• One logical change, one check-in.

Chapter 5 - Reflect
"The six stages of debugging" and reads as follows:
1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why is that happening?
5. Oh, I see.
6. How did that ever work?

• Take the time to perform a root cause analysis:
– At what point in your process did the error arise?

– What went wrong?
• Ensure that the same problem can’t happen again:
– Automatically check for problems.
– Refactor code to remove the opportunity for incorrect usage.
– Talk to your colleagues, and modify your process if appropriate.
• Close the loop with other stakeholders.

Chapter 6 - Discovering that you have a problem
• Make the most of your bug-tracking system:
– Pick one at an appropriate level of complexity for your particular situation.
– Make it directly available to your users.
– Automate environment and configuration reporting to ensure accurate reports.
• Aim for bug reports that are the following:
– Specific
– Unambiguous
– Detailed
– Minimal
– Unique
• When working with users, do the following:
– Streamline the bug-reporting process as much as possible.
– Communication is key—be patient and imagine yourself in the user’s shoes.
• Foster a good relationship with customer support and QA so you can leverage their support during bug fixing.

Chapter 7 - Pragmatic Zero Tolerance
• Detect bugs as early as possible, and fix them as soon as they come to light.
• Act as though bug-free software was an attainable goal, but temper perfectionism with pragmatism.
• If you find yourself faced with a poor quality codebase, do the following:
– Recognize there is no silver bullet.
– Make sure that the basics are in place first.
– Separate clean code from unclean, and keep it clean.
– Use bug triage to keep on top of your bug database.
– Incrementally clean up bad code by adding tests and refactoring.

Chapter 8 - Special Cases

• When patching an existing release, concentrate on reducing risk.
• Keep on the lookout for compatibility implications when fixing bugs.
• Ensure that you have completely closed any timing windows, not just decreased their size.
• When faced with a heisenbug, minimize the side effects of collecting information.
• Fixing performance bugs always starts with an accurate profile.
• Even the most restricted communication channel can be enough to extract the information you need.
• Suspect your own, ahead of third-party, code.

Chapter 9 - The Ideal Debugging Environment
• Automate your tests, ensuring that they do the following:
– Unambiguously pass or fail
– Are self-contained
– Can be executed with a single click

– Provide comprehensive coverage
• Use branches in source control sparingly.
• Automate your build process:
– Build and test the software every time it changes.
– Integrate static analysis into every build.

Chapter 10 - Teach your Software to Debug Itself
• Use assertions to do the following:
– Both document and automatically validate your assumptions
– Ensure that your software, although robust in production, is fragile during debugging
• Create a debug build that
– Is compiled with debug-friendly compiler options
– Allows key subsystems to be replaced by debugging equivalents
– Builds in control that will prove useful during diagnosis
• Detect systemic problems, such as resource leaks and exception handling issues, preemptively.

Chapter 11 - Anti Patterns
• Keep on top of your bug database to ensure that it accurately reflects your true priorities.
• The polluter pays—don’t allow anyone to move onto a new task until they’ve completely finished their current one. If bugs come to light in their work, they fix them.
• Make a single team responsible for a product from its initial concept through deployment and beyond.
• Firefighting will never fix a quality problem. Take the time to identify and fix the root cause.
• Avoid “big bang” rewrites.
• Ensure that your code ownership strategy is clear.
• Treat anything you don’t understand as a bug.

Nikita Salnikov-tarnovski says

I love this book. I really wish every developer would read it. I have seen way too often in my career that developers cannot systematically debug their own code. This book gives very good methodology to follow, describe the correct setting and common pitfalls. Will certainly recommend it to my colleagues.

The only critique I have is about performance-related chapter. Being a performance engineer myself I would recommend to skip this chapter entirely :)

Anton Antonov says

The book is like a press release. 30 pages of info, spread into 200 pages, leaving you to wonder why did you read the whole thing.

Really, the book has nothing in-depth to offer. Just the most basic of basic advices. The last anti-pattern section might slightly offer something new, but that's all.

Huy Tran says

If you're an experienced developer, you might already know almost everything the book is talking about, but it's interesting to see the them in an organized way and written down.

Erika RS says

This book was valuable but pretty basic. Someone without a lot of debugging experience could learn a lot of useful tips and ways of thinking about debugging. One key lesson is that the purpose of debugging is to find the cause of an issue, *not* to fix the bug. If you fix a bug without understanding what caused it, then you've failed at debugging. For the more experienced debugger, there will only be one or two useful things. That said, it's a quick enough read to be worth reading for those one or two things.

I really like the idea of applying a root cause analysis to every bug. This goes beyond figuring out why the code caused the particular issue and into analyzing why the code got that way in the first place. Is there a systematic problem with the code? Is the code health in the relevant module so bad that bugs are easy to let in? Was the code developed under rushed conditions? Debugging is an opportunity not just to improve code, but to improve your overall development process.

There was also a good section on digging yourself out of a quality hole in a system with too many issues. The two high level techniques are keep things from getting worse and tackle the problematic areas of the code, but putting this sort of quality focus in the context of debugging really helps to broaden the picture.

The final chapter on anti-patterns, made me cringe with recognition several times. In particular, priority inflation (only high priority bugs are addressed, so all bugs are given high priority), firefighting (the team is so busy doing that they never step back and analyze how to make things better), and no code ownership (no one feels responsibility for a particular module, and so bugs in it don't get fixed). These anti-patterns all have solutions, but there is no silver bullet. Much of it comes back to discipline and prioritization.

All-in-all, this book is worth the few hours it takes to read.

David MacIver says

I liked the exposition in this book, and generally agree with the principles it espouses. It didn't really teach me much, but it might have when I was earlier in my career. I've passed my copy on to my brother who is rather newer to programming. Hopefully he'll find it useful.

Tim says

While some of the tips may seem obvious to those with even a few years experience (you do use some form of source control, don't you?), Paul not only helps find bugs after they have occurred but also before with tips about automated testing, the use of asserts, logging, etc. Reading all the obvious tips written down in one place helped me remember that all is not lost when trying to debug the thorniest of issues. While many of the

of the code examples are Java, other languages are not forgotten. There is some Python, Ruby, and C and C++.

In the section on Anti-Patterns, Paul gives advice and tips on office politics, for example dealing with Prima Donnas and issues of code ownership. The tools section covers a wide variety, not just focused on digging out bugs, bug tracking them and, just in case you don't, source control.

The book is written in a very easy going style and I can imagine a talk by him being very well received at conferences like ACCU.

Disclaimer: I was provided with a copy of the book to review.

Vasil Kolev says

Finally, there's a book about debugging.

Except part two, the rest of the book is great, has almost all the advice needed for debugging properly, and although it's mostly written for software developers, a lot of it can be used in network or system debugging. It's the kind of book every programmer should read at least once.

James Anderson says

This book changed my career! I read this during my year of placement at a company. My job was to make bug fixes and small features. I spent 3 days on this particular bug and that’s a lot of time and money for the customer. I decided I must be taking the wrong approach. I read half of this book over the weekend and had the bug solved and resolved by the end of Monday.

This made me think of software development as my craft.

Ashraf Bashir says

The book contains perfect tips for beginners and professionals. It also contains many life situations which serves as good examples. The first part is much powerful and well descriptive. It can be considered a "debugging bible". But the second part is a bit subjective, and didn't cover each topic in details. I think the second part should be refined with more life examples and code samples, and it should be reconstructed to be more organized. But, overall, the book is great, and the author knows how to keep the user reading without getting bored. I advise all developers (and also testers!) to read it and practice its suggestions. BTW, the last chapter contains some tools which are very useful, you may skim through some of these tools and get used to them, they are very useful. This book is most recommended specially the first part !

Stefan Kanev says

This book was an interesting read. It has the quality of being simple on the surface, but once you start thinking deeply about it, there are a lot of fine points to learn. You can call a lot of it "common sense", but it's the insidious type of common sense that most people know should follow, but seldom do.

Probably the main thing to grok is that debugging is not tools, but a mental process. Shiny software is fun, but in most cases, printf is all you need. The difference between effective and ineffective debugging is not what tools you use, but how you structure your approach and think about the problem. That's what this book can teach if you pay attention.

If you struggle with debugging, this book is definitely for you. You should read it carefully and take all in, even if it seems like common sense. I fancy myself very good at debugging, but I still learned a few tricks here and there.

Here's a subset of the things I've jotted down when reading it:

• There are always four steps and you should never skip one: Reproduce, Diagnose, Fix, Reflect.
• On the previous note, don't forget to start with reproduction.
• It's important to have a minimal feedback loop when debugging.
• It's also important to look for a minimal reproduction – when 10 steps reproduce the bug, see if you can do it with less.
• Experiments must prove something. Don't forget to have a notion about what you intend to prove.
• Keep a record of things you've tried. Very useful for long debugging session.
• "The key is to understand what assumptions you’re making, as well as when to examine them critically. A particularly good time to do so is when you’re stuck – it may be because one of them is blinding you to what’s really going on."
• Get to the root cause – don’t just make the problem go away (like function being called with larger array), but figure out what is wrong upstream and address that.
• You should be either modifying behaviour or refactoring, but never at the same time.
• To avoid problems, it helps to stick to one change at a time. Making small commits help with that.
• The six stages of debugging: (1) This can’t happen, (2) That doesn’t happen on my machine, (3) That shouldn’t happen, (4) Why is that happening?, (5) Oh, I see, (6) How did that ever work?
• Before moving to a bug, consider whether the original mistake was a one-off. It might be that the original author doesn’t understand something generic. Go and get involved in this.

????? ????? says

? ???? ?????? ? ???? ???????? ??????? ????????? ????????.

? ????, ??? ????? ????? ??? ??, ??? ?????????? ?????????, ??? ?????? ?????? ? ?????-?? ??????? ???????, ??? ????????? ? ????? ???????????????????? (?????? ??? ?????, ????? ??? ????????? ??? ???????????). ?????, ? ????????, ??? ??? ??? ????, ?? ??????? ?????? ??? ? ????. ??? ??????????? ??????? ???????? ????? ??? ????? ?? ???????????. ???, ???????, 60 ??????? ?? 200, ?? ??? ????? ??????? ????.

???? ????? ??? ??, ??? ??????. ?????? ???????????????? ????????????, CI ? ?????? ??????. ??? ?????

???????????????? ?? ???? ????, ?????? ?????? ??????? ? ??? ??? ?????? ??? ????????. ???? ?????-?? ??????, ??? ????????? ?? ???????????? ???????? ???? "??? ??????, ???? ??? ???????? ????????". ??? ??? ????????? ???????????? ??????. ?? ??? ?????? ??? ??????????, ??????? ?????? ????? ?????????????. ?? ????? ????????? ???????????, ??? ????? ?????? ??? ?? ????? ??????? ??????. ? ????? ???????????? ?????????? ??? ???, ??? ??????.

?? ???? ???????, ??? ????? ???? ?? ?????? ??????. ??????, ??????? ??????, ?????? ???? ???????????? (???? ?????????, ???????, ???????). ????? ?? Pragmatic Bookshelf, ???????????????? ????? (????? ???????????-??????????????), ? ?????????????? ???????? ?? Pragmatic Programmer.

?? ????????????, ??? ???? ?????? ???????????, -- ? 10 ????? ?????????? ?????? ???????????? ?++ ??????? ????????????, ??????? ?? ??? ????? ????????. ???????????? ????? ????????? ? ???????????? ??????? ??????: ?? ?????? ??????? ???????????? ? ????? ??????? ?? ????????, ? ?? ?????? ??????? ??? ?????????? ???????? ?? ??????? ?????????????. ???????????? ????? ???????????? ??? ???????????? ????????? ????????. ?? ? ????? ????????? ????????? ??? ????????? ?????????? ? ???-?? ?????????, ??? ??? ???????? ????????? ???????, ????? ??????????.

?????, ????????????? ?? ?????. ????? ?????????? Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems.

David says

As with so many software engineering books, there's a fair bit to like in this book, and some to hate. Generally, I'm a fan of the pragmatic approach. Many of the principles in this book feel broadly useful in debugging, and the author emphasizes real understanding, solving the right problems, and verifying that you've solved the right problems. That said, this book has its own dogmas (e.g., on reproducibility) or other advice (e.g., on assertions in production) that align with what I think are frustrating and counter-productive industry anti-patterns.

Inexperienced engineers could probably learn a lot from the ideas in this book, but I can't tell whether inexperienced engineers _would_ learn it from a book like this. Much of the content is the kind of stuff that's hard to appreciate until you've done the wrong thing and see why the suggested thing is better. I learned much of this through experience and through the mentorship of experienced engineers, but I hope people can also develop on their own through books like this.

Kristjan Wager says

While it's probably a quite good introduction to the field of debugging, even for experienced programmers, I found it a bit too basic for those of us who are more experienced with debugging.

The book seemed to lack a clear focus on what its target group was - some parts of it were clearly aimed towards the individual programmer, while other parts of it were aimed towards teams or project leaders. Unfortunately, these parts were mixed together (sometimes within the same paragraph).

Speaking as a .NET developer, I also found the book strangely focused on Java and C/C++. This is probably

due to the expertise of the author, but it would have been nice if all the tools (and problems) mentioned hadn't been solely within those domains.
