WINNER OF JOLT PRODUCTIVITY AWARD

ANDREAS ZELLER

WHY PROGRAMS FAIL

A GUIDE TO SYSTEMATIC DEBUGGING

SECOND EDITION

dpunkt.verlag

MORGAN KAUFMANN

Why Programs Fail
A Guide to Systematic Debugging

Second Edition

Andreas Zeller

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Morgan Kaufmann Publishers is an imprint of Elsevier
30 Corporate Drive, Suite 400
Burlington, MA 01803

This book is printed on acid-free paper. ♾

Library of Congress Cataloging-in-Publication Data
Application submitted.

ISBN-13: 978-0-12-374515-6

Printed in the United States
08 09 10 11 12   10 9 8 7 6 5 4 3 2 1

Why Programs Fail

Foreword

In *Federalist 51*, James Madison wrote: "If men were angels, no government would be necessary." If he lived today, Madison might have written: "If software developers were angels, debugging would be unnecessary." Most of us, however, make mistakes, and many of us even make errors while designing and writing software. Our mistakes need to be found and fixed, an activity called debugging that originated with the first computer programs. Today every computer program written is also debugged, but debugging is not a widely studied or taught skill. Few books, beyond this one, present a systematic approach to finding and fixing programming errors.

Be honest: Does debugging seem as important, difficult, or worthy of study as writing a program in the first place? Or, is it just one of those things that you need to do to finish a project? Software developers though spend huge amounts of time debugging—estimates range up to half or more of their workdays. Finding and fixing bugs faster and more effectively directly increases productivity and can improve program quality by eliminating more defects with available resources. Preventing mistakes in the first place would be even better, but no one has yet found the technique to prevent errors, so effective debugging will remain essential.

Improved programming languages and tools can supplant, but not eliminate, debugging, by statically identifying errors and by dynamically detecting invariant violations. For example, the type system in modern languages such as Java and C# prevents many simple mistakes that slip by C programmers. Moreover, these languages' runtime bounds checks stop a program when it strays out of bounds, which may be billions of instructions before the error manifests itself. Unfortunately, there are countless ways in which a program can go wrong, almost all of which languages and tools cannot detect or prevent. For example, in recent years there has been considerable work in verifying sequences of operations in a program. Tools can ensure that a file is opened before a program reads it, but they cannot check that the correct file is accessed or that the program properly interprets its contents. If either mistake occurs, someone still must debug the program to understand the error and determine how to fix it.

In addition, debugging can be an enjoyable activity that shares the thrill of the hunt and chase found in a good detective novel or video game. On the other hand, a protracted, unsuccessful search for a bug in your code quickly loses its charm, particularly when your boss is asking repeatedly about your (lack of) progress. Learning to debug well is essential to enjoying software development.

This book can teach you how to debug more effectively. It is a complete and pragmatic overview of debugging, written by a talented researcher who has developed many clever ways to isolate bugs. It explains best practices for finding and fixing errors in programs, ranging from systematically tracking error reports, reproducing failures, observing symptoms, isolating the cause, and correcting defects.

Along with basic techniques and commonly used tools, the book also explores the author's innovative techniques for isolating minimal input to reproduce an error and for tracking cause and effect through a program.

Studying this book will make you a better programmer. You will be able to find and fix errors in your code (and your colleague's code) faster and more effectively, a valuable skill that will enable you to finish projects earlier and produce programs with fewer defects. Also, if you read between the lines you will learn how to write code that is more easily tested and debugged, which further increases your ability to find and correct defects. And thinking hard about what can go wrong with your program can help you avoid mistakes in the first place, so you have less to debug.

James Larus
Microsoft Research

Preface

This is a book about bugs in computer programs—how to reproduce them, how to find them, and how to fix them such that they do not occur. This book teaches a number of techniques that allow you to debug any program in a systematic, and sometimes even elegant, way. Moreover, the techniques can be widely automated, which allows you to let your computer do most of the debugging. Questions this book addresses include:

- How can I reproduce failures faithfully?
- How can I isolate what is relevant for the failure?
- How does the failure come to be?
- How can I fix the program in the best possible way?
- How can I learn from mistakes to prevent future ones?

Once you understand how debugging works, you will not think about debugging in the same way. Instead of seeing a wild mess of code, you will think about causes and effects, and you will systematically set up and refine hypotheses to track failure causes. Your insights may even make you set up your own automated debugging tool. All of this allows you to spend less time on debugging, which is why you are interested in automated debugging in the first place, right?

How This Book Came to Be Written

Although I work as a researcher, I have always considered myself a programmer, because this is how I spend most of my time. During programming, I make mistakes, and I have to debug my code. I would like to say that I am some type of *überprogrammer*—that I never make mistakes—but I am only human, just like anyone else.

During my studies, I have learned that an ounce of prevention is more than worth a pound of cure. I have learned many ways of preventing mistakes. Today, I teach them to my students. However, in striving for prevention we must not forget about the cure. If we were doctors, we could not simply refuse treatment just because our patient had not taken all possible preventive measures.

So, rather than designing yet another ultimate prevention, I have sought good cures. This same pragmatic attitude has been adopted by other researchers around the globe. I am happy to report that we have succeeded. Today, a number of advanced debugging techniques are available that widely *automate the debugging process.*

These techniques not only automate debugging, but also turn debugging from a black art into a systematic and well-organized discipline that can be taught just like any software engineering topic. Thus, I created a course on automated debugging and reworked the lecture notes into a book. The result is what you see before you.

What Is New in this Second Edition

In the past three years, the field of automated debugging has made tremendous advances. This second edition treats some of the most exciting novelties.

A new chapter on "Learning from Mistakes." In Chapter 16, I describe recent work on leveraging change and bug databases, to detect automatically where previous defects were located, and how to predict where the next ones will be.

New insights on how to report problems. Chapter 2 now includes insights from a ground-breaking study by Bettenburg et al., who have surveyed what developers need most in a problem report.

Reproducing crashes. In Chapter 4, I present the *Cdd* and *ReCrash* tools, which allow for automatic reproduction of crashes while requiring little to no overhead. This addresses one of the most pressing problems in debugging.

New material on tracking origins. Chapter 9 now discusses the WHYLINE tool for JAVA, allowing expert developers to ask questions on why specific things happened during execution, or why they did not (e.g., "Why did this error message occur?").

Updated and extended discussions all over the book. Along with several updates on the state of the art, I have also fixed all errors reported by readers, and revised and updated all the material.

Despite the additions, the numbering of chapters, sections, and exercises is virtually unchanged. Thus, references to items in the first edition should also apply to this second edition (which is helpful if you use this book in a course).

Audience

This book is intended for computer professionals, graduate students, and advanced undergraduates who want to learn how to debug programs systematically and with automated support. The reader is assumed to be familiar with programming and manual testing, either from introductory courses or work experience.

What This Book Is and What It Is Not

This book focuses on the *cure* of bugs—that is, the act of isolating and fixing the defect in the program code once a failure has occurred. It only partially covers *preventing* defects. Many other books are available that provide an in-depth treatment of this topic. In fact, one might say that most of computer science is concerned with preventing bugs. However, when prevention fails, there is need for a cure, and that is what this book is about.

Overview of Content

This book is divided into 16 chapters and an Appendix. Chapters 1, 6, and 12 are prerequisites for later chapters.

At the end of each chapter, you will find a section called "Concepts," which summarizes the *key concepts* of the chapter. Some of these concepts are denoted "How To." These summarize *recipes* that can be easily followed (they are also listed in the Contents). Furthermore, each chapter ends with practical *exercises*, for verifying your knowledge, and a "Further Reading" section. This book is organized as follows.

Chapter 1: How Failures Come To Be

Your program fails. How can this be? The answer is that the programmer created a defect in the code. When the code is executed, the defect causes an infection in the program state, which later becomes visible as a failure. To find the defect, one must reason backward, starting with the failure. This chapter defines the essential concepts when talking about debugging, and hints at the techniques discussed subsequently—hopefully whetting your appetite for the remainder of this book.

Chapter 2: Tracking Problems

This chapter deals with the issue of how to *manage* problems as reported by users—how to track and manage problem reports, how to organize the debugging process, and how to keep track of multiple versions. This information constitutes the basic framework in which debugging takes place.

Chapter 3: Making Programs Fail

Before a program can be debugged, we must set it up such that it can be *tested*—that is, executed with the intent to make it fail. In this chapter, we review basic testing techniques, with a special focus on automation and isolation.

Chapter 4: Reproducing Problems

The first step in debugging is to *reproduce* the problem in question—that is, to create a test case that causes the program to fail in the specified way. The first reason is to bring it under control, such that it can be observed. The second reason is to verify the success of the fix. This chapter discusses typical strategies for reproducing an operating environment, including its history and problem symptoms.

Chapter 5: Simplifying Problems

Once we have reproduced a problem, we must *simplify* it—that is, we must find out which circumstances are not relevant to the problem and can thus be omitted. This process results in a test case that contains only the relevant circumstances. In the best case, a simplified test case report immediately pinpoints the defect. We introduce *delta debugging*, an automated debugging method that simplifies test cases automatically.

Chapter 6: Scientific Debugging

Once we have reproduced and simplified a problem, we must understand how the failure came to be. The process of arriving at a theory that explains some aspect of the universe is known as *scientific method*. It is the appropriate process for obtaining problem diagnostics. We introduce basic techniques of creating and verifying hypotheses, creating experiments, conducting the process in a systematic fashion, and making the debugging process explicit.

Chapter 7: Deducing Errors

In this chapter, we begin exploring the techniques for creating hypotheses that were introduced in Chapter 6. We start with *deduction* techniques—reasoning from the *abstract* program code to the *concrete* program run. In particular, we present *program slicing*, an automated means of determining possible origins of a variable value. Using program slicing, one can effectively narrow down the number of possible infection sites.

Chapter 8: Observing Facts

Although deduction techniques do not take concrete runs into account, observation determines *facts* about what has happened in a concrete run. In this chapter, we look under the hood of the actual program execution and introduce widespread techniques for examining program executions and program states. These techniques include classical logging, interactive debuggers, and postmortem debugging—as well as eye-opening visualization and summarization techniques.

Chapter 9: Tracking Origins

Once we have observed an infection during debugging, we need to determine its origin. We discuss *omniscient debugging*, a technique that records an entire execution history such that the user can explore arbitrary moments in time without ever restarting the program. Furthermore, we explore *dynamic slicing*, a technique that tracks the origins of specific values.

Chapter 10: Asserting Expectations

Observation alone is not enough for debugging. One must *compare* the observed facts with the expected program behavior. In this chapter, we discuss how to automate such comparisons using well-known *assertion* techniques. We also show how to ensure the correct state of important system components such as memory.

Chapter 11: Detecting Anomalies

Although a single program run can tell you quite a bit, performing multiple runs for purpose of comparison offers several opportunities for locating *commonalities* and *anomalies*—anomalies that frequently help locate defects. In this chapter, we discuss how to detect anomalies in code coverage and anomalies in data accesses. We also show how to infer invariants from multiple test runs automatically, in order to flag later invariant violations. All of these anomalies are good candidates for identification as infection sites.

Chapter 12: Causes and Effects

Deduction, observation, and induction are all useful in finding *potential* defects. However, none of these techniques alone is sufficient in determining a *failure cause*. How does one identify a cause? How does one isolate not just *a* cause but *the* actual cause of a failure? This chapter lays the groundwork for techniques aimed at locating failure causes systematically and automatically.

Chapter 13: Isolating Failure Causes

This chapter is central to automating most of debugging. We show how delta debugging isolates failure causes automatically—in program input, in the program's thread schedule, and in program code. In the best case, the reported causes immediately pinpoint the defect.

Chapter 14: Isolating Cause–Effect Chains

This chapter presents a method of narrowing down failure causes even further. By extracting and comparing program states, delta debugging automatically isolates the *variables and values* that cause the failure, resulting in a cause-effect chain of the failure: For example, "variable `x` was 42; therefore `p` became null, and thus the program failed."

Chapter 15: Fixing the Defect

Once we have understood the failure's cause-effect chain, we know how the failure came to be. However, we must still locate the origin of the infection—that is, the actual location of the defect. In this chapter, we discuss how to narrow down the defect systematically—and, having found the defect, how to fix it.

Chapter 16: Learning from Mistakes

At the end of each debugging session, one wonders how the defect could have come to be in the first place. We discuss techniques to collect, aggregate, and locate defect information; techniques to predict where the next defects will be; and what to do to prevent future errors.

Appendix: Formal Definitions

For the sake of readability, all formal definitions and proofs have been grouped in the Appendix.

Glossary

The Glossary defines important terms used throughout the book.

Bibliography

The bibliography presents a wide range of sources of further reading in the topics covered by the text.

Supplements, Resources, and Web Extensions

Much of the material covered in this book has never been discussed in a textbook before. The later chapters have not been widely tested in practice, and like any book on an evolving field, this one will benefit from more refinement and from further work. In other words, this book is full of bugs, and I welcome any comments on it. You can write to me care of Morgan Kaufmann, or email me at zeller@whyprogramsfail.com. There is also a Web page at *http://www.whyprogramsfail.com* for late-breaking information and updates (read: errata).

Advice for Instructors

I have used this book for five graduate courses on automated debugging. Each course consisted of approximately 16 lectures of 60 to 90 minutes each. Essentially, there was one lecture per chapter. The exercises stem from these courses (and their exams). I have also used parts of the book for a number of tutorials on debugging, as well as for inclusion in programming and software engineering courses. For your convenience, my presentation slides for these courses are available in Keynote and Powerpoint format. Instructions on how to access them are available at *http://www.whyprogramsfail.com*.

If you prefer to make your own slides, all of the original illustrations for this book are also available at this site.

Advice for Readers

Typographics

To keep things simple, most examples in this book use simple input/output mechanisms—that is, the *command line* and the *console*. In all of these examples, `typewriter font` stands for program output, and **`bold typewriter font`** for user input. The command-line prompt is denoted by a dollar sign ($), and the cursor by an underscore (_). The following is a simple example. The user invokes the `hello` program, which prints the text `Hello, world!` on the console.

```
$ ./hello
Hello, world!
$ _
```

Programming Environment

The concepts and techniques discussed in this book do not depend on a particular programming environment or operating system. To illustrate the techniques, though, I frequently use *command-line tools*, typically from the Linux/UNIX community. In addition to saving space, this is a matter of simplicity; these command-line tools provide a functional core similar to that found in almost all sophisticated programming environments. Therefore, you should have no trouble transferring the examples to your personal programming workbench.

ACKNOWLEDGMENTS

Many people have had a part in the development of this book. I would like to thank everybody who reviewed drafts of the manuscript or parts of it: Philipp Bouillon, Holger Cleve, David Cok, Michael Ernst, David Evans, Clint Jeffery, Dieter Kranzlmüller, Jens Krinke, Raimondas Lencevicius, Bil Lewis, Ben Liblit, Christian Lindig, Edu Metz, Robert Muth, Stephan Neuhaus, Jeff Offutt, Kerstin Reese, Zoltan Somogyi, Peter Weißgerber, Thomas Zimmermann, and the students of the *Automated Debugging* courses at Saarland University, the University of Washington, and ETH Zürich. Philipp Bouillon, Silvia Breu, Holger Cleve, Andreas Leitner, and Martin Mehlmann also helped with the conception of the exercises. Thanks to all readers who reported bugs in the first edition, in particular Martin Burger, Roland Illig, and Yong-Wu Wang. Christa Preisendanz of dpunkt Verlag and Tim Cox of Morgan Kaufmann Publishers were most helpful. And finally, my family has supported me enormously while writing this book—it's great to have you with me.

Have fun debugging!

For the Snark's a peculiar creature, that won't
Be caught in a commonplace way.
Do all that you know, and try all that you don't:
Not a chance must be wasted to-day!

— LEWIS CARROLL
The Hunting of the Snark (1876)

Contents