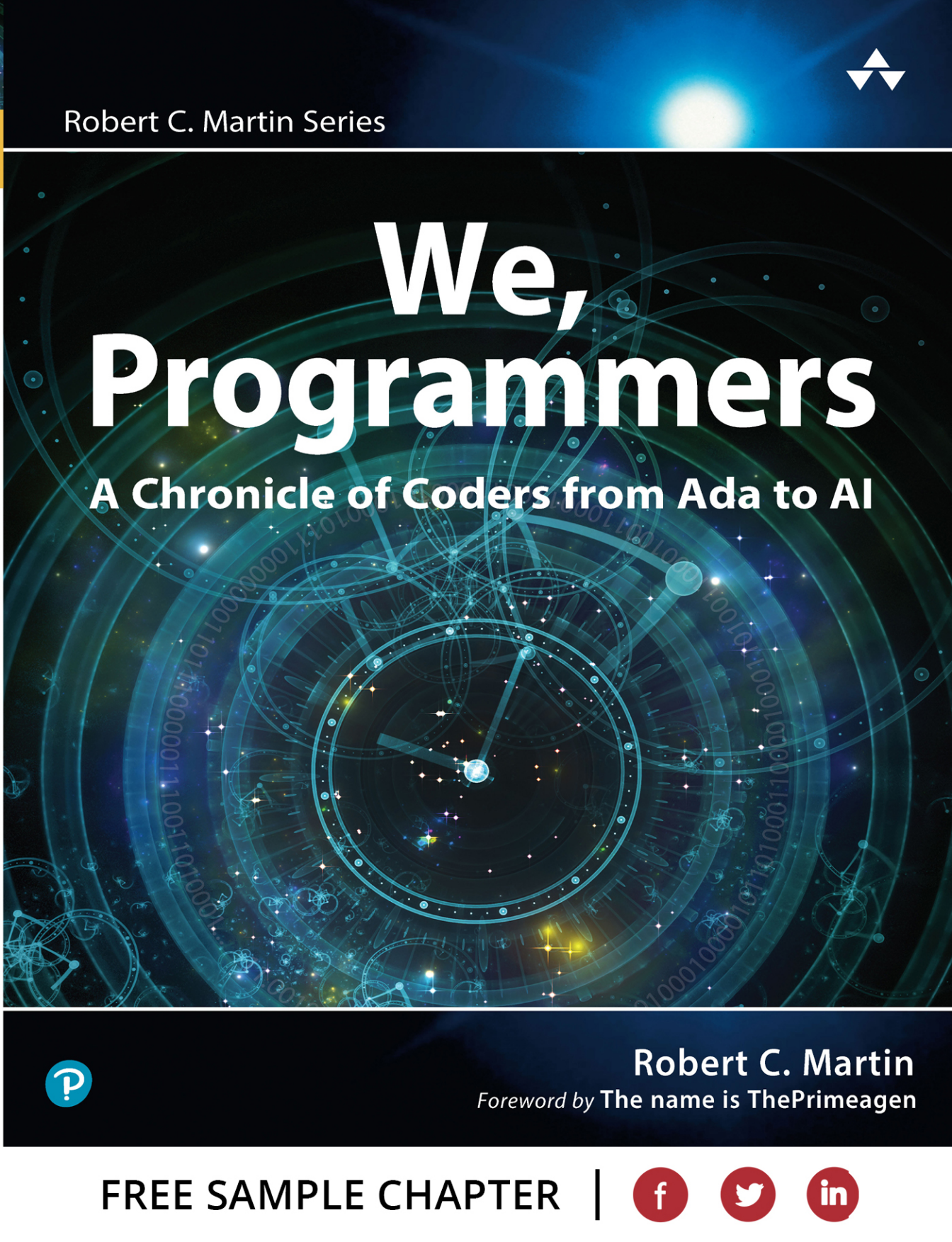

Robert C. Martin Series
We, Programmers
A Chronicle of Coders from Ada to AI
Robert C. Martin
Foreword by The name is ThePrimeagen
FREE SAMPLE CHAPTER

Praise for We, Programmers

"I, like Uncle Bob, spent much of my career consulting, teaching, and going to computer conferences. The importance of this is that I got to meet and dine with many of the characters in this book. So this book is about my professional friends, and I can tell you that it is a faithful story. In fact, it is incredibly well-written and researched. That is how it *really* was."

—From the Afterword by Tom Gilb

"I can't think of any other book that provides such a sweeping overview of the early history of programming."

—Mark Seeman

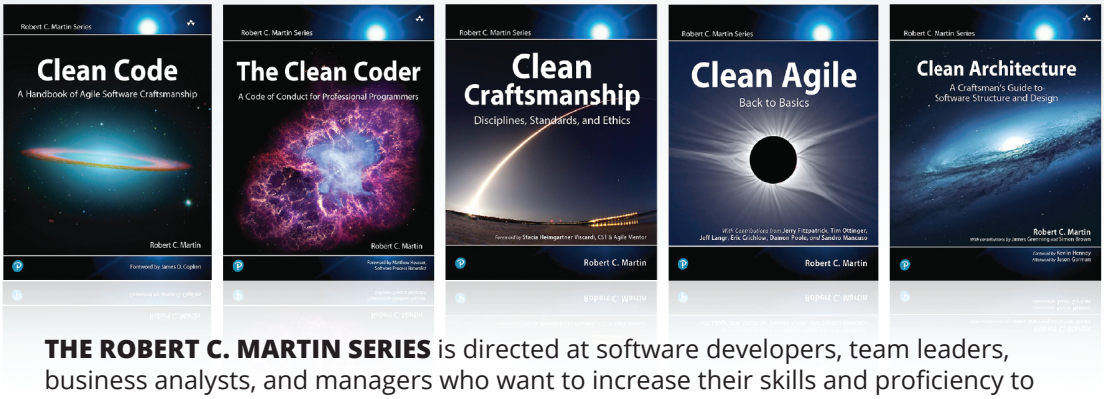
"*We, Programmers* is a fascinating romp through the history of computers and programming. Wonderful glimpses into the lives of some of the greats. And an enjoyable ride with Uncle Bob's career as a programmer."

—Jon Kern, co-author of the Agile Manifesto

"In *We, Programmers*, Bob successfully weaves together a highly entertaining history of programmers, giving us a wealth of historical context, humanizing stories, and eye-opening revelations about the foundational folks in our industry, all bolstered with just the right amount of low-level detail. Bob, being a small piece of this rich history, finds a way to pepper the history with his own relevant observations and critiques. We also get Bob's full story this time out, as well as his own thoughts about what's coming next. A fun, quick read."

—Jeff Langr

Robert C. Martin Series



THE ROBERT C. MARTIN SERIES is directed at software developers, team leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman.

Clean Code presents the principles, patterns, and practices of writing clean code and challenges programmers to closely read code, discovering what's right and what's wrong with it.

Clean Coder provides practical guidance on how to approach software development and work well and clean with a software team.

Clean Craftsmanship picks up where *Clean Code* leaves off, outlining additional ways to write quality and trusted code you can be proud of every day.

Clean Agile is a clear and concise guide to basic Agile values and principles. Perfect for those new to Agile methods and long-time developers who want to simplify approaches for the better.

Clean Architecture brings the methods and tactics of "clean coding" to system design.

Visit **informit.com/martinseries** for a complete list of available publications.

Make sure to connect with us!
InformIT.com/connect

Pearson

informIT

We, Programmers

A Chronicle of Coders from Ada to AI

Robert C. Martin

Addison-Wesley

Hoboken, New Jersey

Cover image: agsandrew/Shutterstock
Page xxxi: Author photo courtesy of Robert C. Martin
Page 31: Jacquard Loom, gorosan/Shutterstock
Page 46: David Hilbert, INTERFOTO/Alamy Stock Photo
Page 53: Hilbert's tomb, Kassandro, licensed under CC BY-SA 3.0
Page 82: Harvard Mark I, Arnold Reinhold, licensed under CC BY-SA 3.0
Page 116: John Backus, courtesy of Lawrence Berkeley National Laboratory
Page 119: SSEC mainframe computer, Everett Collection/Shutterstock
Page 139: ARRA, Internationaal Instituut voor Sociale Geschiedenis (IISG), licensed under CC BY-SA 2.0
Page 147: ARMAC, Gabriele Sowada, licensed under CC BY 3.0
Page 210: Newspaper article with photo of Judith Allen; *The Oregon Journal*'s archive is the property of Oregonian Publishing Co.
Page 213: Second gen computers, courtesy of Paul Pierce
Page 224: Operator working with early CAD program, courtesy of the Computer History Museum
Page 245: Digi-Comp, Pierre Terre, licensed under CC BY-SA 3.0
Page 259: Varian 620, J R Spigot, licensed under CC0 1.0
Page 262: Tri-data Cartridge, Museum of Obsolete Media, by Jason Curtis licensed under CC BY SA 4.0
Page 267: System7, Mike Ross/corestore.org
Page 373: Grace Murray Hopper, official U.S. Navy Photograph, from the collections of the Naval History and Heritage Command; photo # NH 96919-KN
Page 392: IBM 026 Keypunch illustration, Jennifer Kohnke
Page 393: IBM Selectric typewriter, Steve lodefink, licensed under CC BY 2.0
Page 393: IBM Selectric Type Ball, David Whidborne/Shutterstock
Page 394: Intel 8080, the Board of Trustees of the Science Museum; image licensed under CC BY-SA 4.0
Page 398: PDP-7, Tore Sinding Bekkedal, licensed under CC BY-SA 1.0
Page 399: PDP-11, Don P. Mitchell/mentallandscape.com
Page 400: Plugboard, Chris Shrigley, licensed under CC BY 2.5
Page 401: Rk07drive, Gunkies.org/Computer History Wiki, licensed under the GNU Free Documentation License 1.2
Page 401: Rk07 Packs, Gunkies.org/Computer History Wiki, licensed under the GNU Free Documentation License 1.2
Page 404: Transistors, Jules Selmes/Pearson Education Ltd.
Page 406: Vacuum tube, bearwu/Shutterstock
Page 407: VT100, Living Computer Museum, Seattle by Jason "Textfiles" Scott, licensed under CC BY 2.0

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024947905

ISBN-13: 978-0-13-534426-2
ISBN-10: 0-13-534426-3

$PrintCode

For Timothy Michael Conrad

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

```
vim .
```

Five simple characters launch my favorite text editor. This isn't just any text editor, it's NeoVim. Today's NeoVim experience sports keybinds, LSPs, syntax highlighting, in-editor error diagnostics, and more. With all of this customization, NeoVim launches in mere milliseconds, allowing file editing in what feels like an instant. Even with thousands of files, LSPs quickly report the state of the project, and errors are loaded into quickfix menus for speedy navigation. A couple keystrokes allow me to build and launch or run tests. My computer produces code from plain English via the AI! Furthermore, that same AI can code alongside me as I type, providing large amounts of (highly questionable) code in a flash. This all sounds *impressive*. The NeoVim experience is wonderful, smooth, and blazingly fast. However, using NeoVim is considered *primitive*, and by some, profane. "Luddite!" some developers shout for choosing to use NeoVim and spending time configuring my editor when full-batteries-included *environments* exist. IntelliJ supports actions that my NeoVim mind cannot even comprehend!

I tell you all of this because it is shocking. No, not shocking that I use what some would call antiquated technologies. Not shocking how software engineers argue over preferences—all that is par for the course. What truly boggles my mind is how the sheer power of editing is mundane to us engineers. A mere blip in the daily grind of coding, meetings, and Slack messages. Text editing is ordinary. Autocompletes, syntax highlighting, reliable (sometimes) docs are just things we expect. Before text editors, engineers spent decades without a high-level language, even longer without syntax highlighting, and practically 70 years without LSPs to give autocomplete and refactoring tools in almost any language. Text editing is truly a marvel of humankind.

Besides living in the past with my text editor, I also love reading about the past. The "real programmers" who could time their code with the drum memory for optimal reading speeds. What I wouldn't give to see one of the greats, experts of their craft, in action. Perhaps it's just nostalgia, but the past adventures seem more grand, discoveries more significant, and work more meaningful. In *We, Programmers*, I got the chance to walk that past, side by side, with the creator(s) of each significant leap in computing history. I could see the dinner parties of Charles Babbage, who inspired and terrified his guests with the Difference Engine. That giant mechanical monster clinking and clanking as it produced what must have appeared as magic. Seeing the Difference Engine in action was likely similar to what we experienced with our first LLM prompt or Copilot autocomplete. I bet you could have heard dinner guests exclaiming "Machines can really think." I got to feel the pressure of teams working around the clock and the urgent need for better computing during the critical calculations of World War II that allowed for the atom bomb. No, they did not have Herman Miller chairs and fancy standing desks. Hell, they didn't even have a monitor or a keyboard! Yet they achieved the unimaginable and changed the course of history. *We, Programmers* is one of the most compelling tellings of computing history.

I would be very surprised to meet a programmer who hasn't heard the name Uncle Bob or isn't familiar with his work. He is absolutely prolific in our industry. For many years, I only knew of Uncle Bob by name, by

Twitter PFP, and his notable works on clean code and Agile. In my mind, he was the Avatar of AbstractBuilderFactory. That all changed when one day we started interacting on Twitter, which resulted in emails, phone calls, and even a podcast. Through these exchanges, my whole perspective changed. Robert C. Martin is so much more than what my university studies lead me to believe. He is pragmatic and willing to make concessions when needed. During and after our podcast, one of the most consistent comments went along the lines of "He laughs and smiles a lot!" It's a testament to his character and a life well lived. He is a genuine Software Engineer and somebody that we can all learn from.

I am personally tired of the countless arguments over whitespace, text editors, and the OOP vs. FP debates that rage across X in 280 characters or less. What I think is interesting is who created the technologies behind the arguments that have shaped so many of us. *We, Programmers* delivers something many times more meaningful, a connection to the past and hope for the future, and Uncle Bob is perhaps the perfect mediator for this story.

—The name is ThePrimeagen

This page intentionally left blank

Preface

I am about to tell you the story of how it all began. It's a twisty-turny tale of the lives and challenges of some remarkable people, the remarkable times in which they lived, and the remarkable machines they mastered.

But before we dive headlong into those twisty little passages all different, a little preview is likely appropriate—just to whet your appetite.

Necessity may be the mother of invention, but nothing breeds necessity like war. The impetus for our industry was created by the paroxysms of war—especially World War II.

In the 1940s, the technology of warfare had outstripped our computational resources. There was simply no way that battalions of humans operating desk calculators could keep up with the computational demands coming from all sectors of the war.

The problem was the vast amount of additions, subtractions, multiplications, and divisions required to approximate the path of a shell fired from a gun to its target. Such problems could not be solved by a simple mathematical formula like $d=rt$ or $s=\frac{1}{2}at^2$. These problems required that time and

distance be broken down into thousands of tiny segments and that the path of the projectile be simulated and approximated from segment to segment. Such simulation requires a vast amount of brute-force grade-school calculation.

In centuries past, all that calculation was performed by armies of humans equipped with pen and paper. It was only in the last century that they were given adding machines to assist in that task. Organizing the calculations, and the teams of people who performed them, was a herculean task.[1] The calculations themselves could take such teams weeks, or even months.

Machines that could perform such feats had been dreamed of in the 1800s. Some anemic prototypes had even been built. But they were playthings and oddities. They were devices to be shown at the dinner parties of the elite. Few considered them to be tools worthy of use—especially considering their cost.

But WWII changed all that. The need was dire. The cost was irrelevant. And so those early dreams became reality, and vast calculating engines were built.

The people who programmed and operated those machines were the pioneers of our field. At first, they were forced into the most primitive of conditions. Programming instructions were literally punched, one hole at a time, into long ribbons of paper tape that the machine would consume and execute. This style of programming was overwhelmingly laborious, hideously detailed, and utterly unforgiving. Moreover, the execution of such programs could span weeks that required detailed monitoring and constant intervention. A loop in a program, for example, was executed by manually repositioning the paper tape for every iteration of the loop, and

1. To see such a task in action, I recommend the 2024 Pi Day celebration, where over 100 digits were calculated by hand by a very well-coordinated team of a few hundred individuals over the period of a week. At the time of writing, "The biggest hand calculation in a century! [Pi Day 2024]", posted to YouTube by Stand-up Maths, Mar. 13, 2024.

manually inspecting the state of the machine to see if the loop should be terminated or not.

As the years wore on, electromechanical machines gave way to electronic vacuum tube machines that stored their data in sound waves traveling through long tubes of mercury. Paper tape gave way to punched cards and eventually to stored programs. These new technologies were driven by those early pioneers, and enabled further innovations.

The first compilers of the early 1950s were little more than assemblers with special keywords that loaded and invoked prewritten subroutines—sometimes from paper tape, or magnetic tape. Later compilers experimented with expressions and data types but remained primitive and slow. By the late '50s, John Backus's FORTRAN and Grace Hopper's COBOL introduced a whole new mindset. The binary code that programmers had previously written by hand could now be generated by a computer program that could read and parse abstract text.

In the early '60s, Dijkstra's ALGOL drove the level of abstraction higher. A few years later Dahl's and Nygaard's SIMULA 67 drove it higher again.

Structured programming and object-oriented programming emerged from those beginnings.

Meanwhile, John Kemeny and crew brought computing to the common person by creating BASIC and time-sharing in 1964. BASIC was a language almost anyone could understand and use. Time-sharing allowed many people to conveniently and simultaneously use a single, expensive computer.

And then came Ken Thompson and Dennis Ritchie, who, in the late '60s and early '70s, blew open the world of software development by creating C and Unix. After that, we were off to the races.

The mainframe computer revolution of the '60s was followed by the minicomputer revolution of the '70s and the microcomputer revolution of

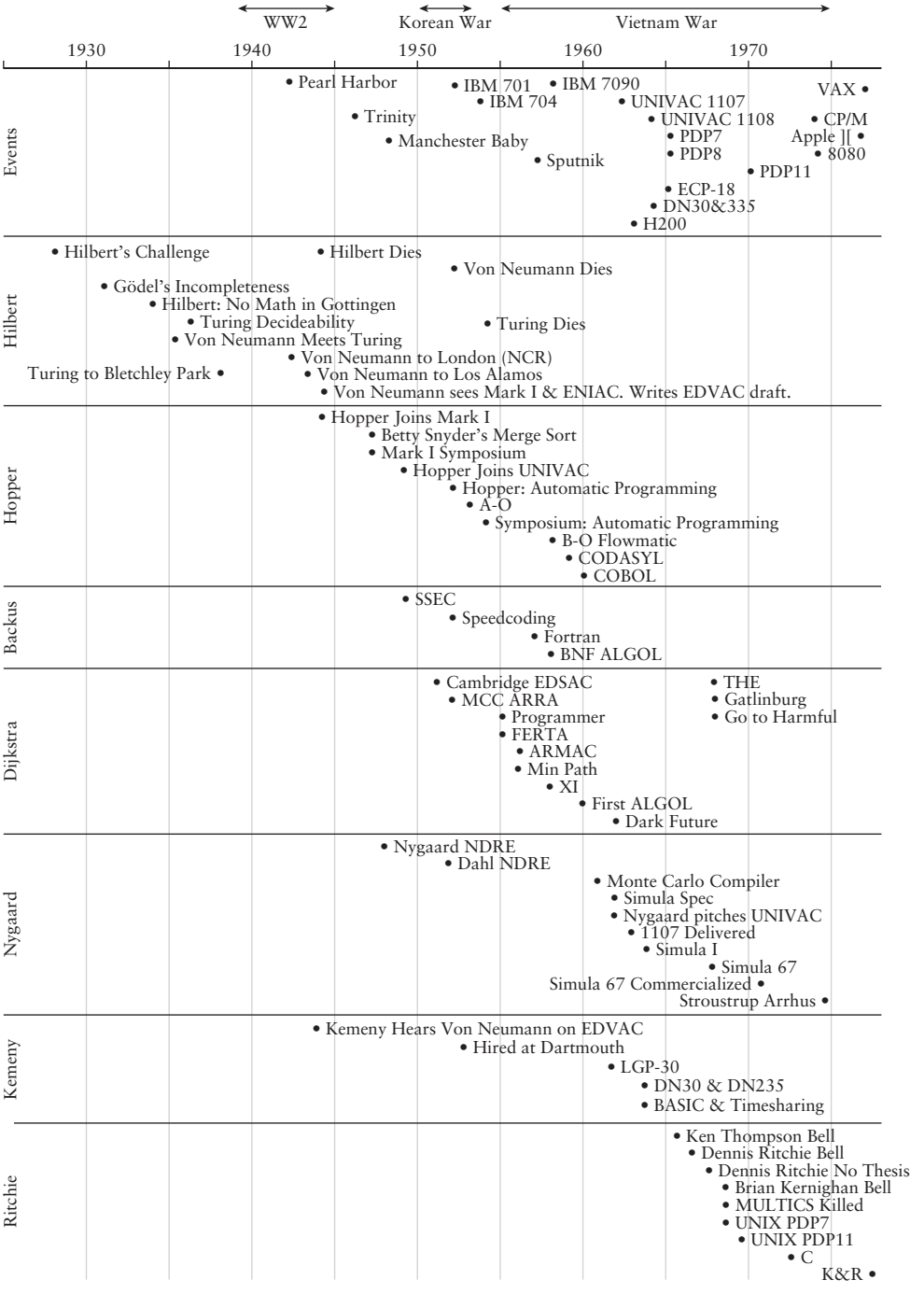
the ’80s. The personal computer took the industry by storm in the ’80s, followed quickly by the object-oriented revolution, and then the internet revolution, and then the Agile revolution. Software was beginning to dominate *everything*.

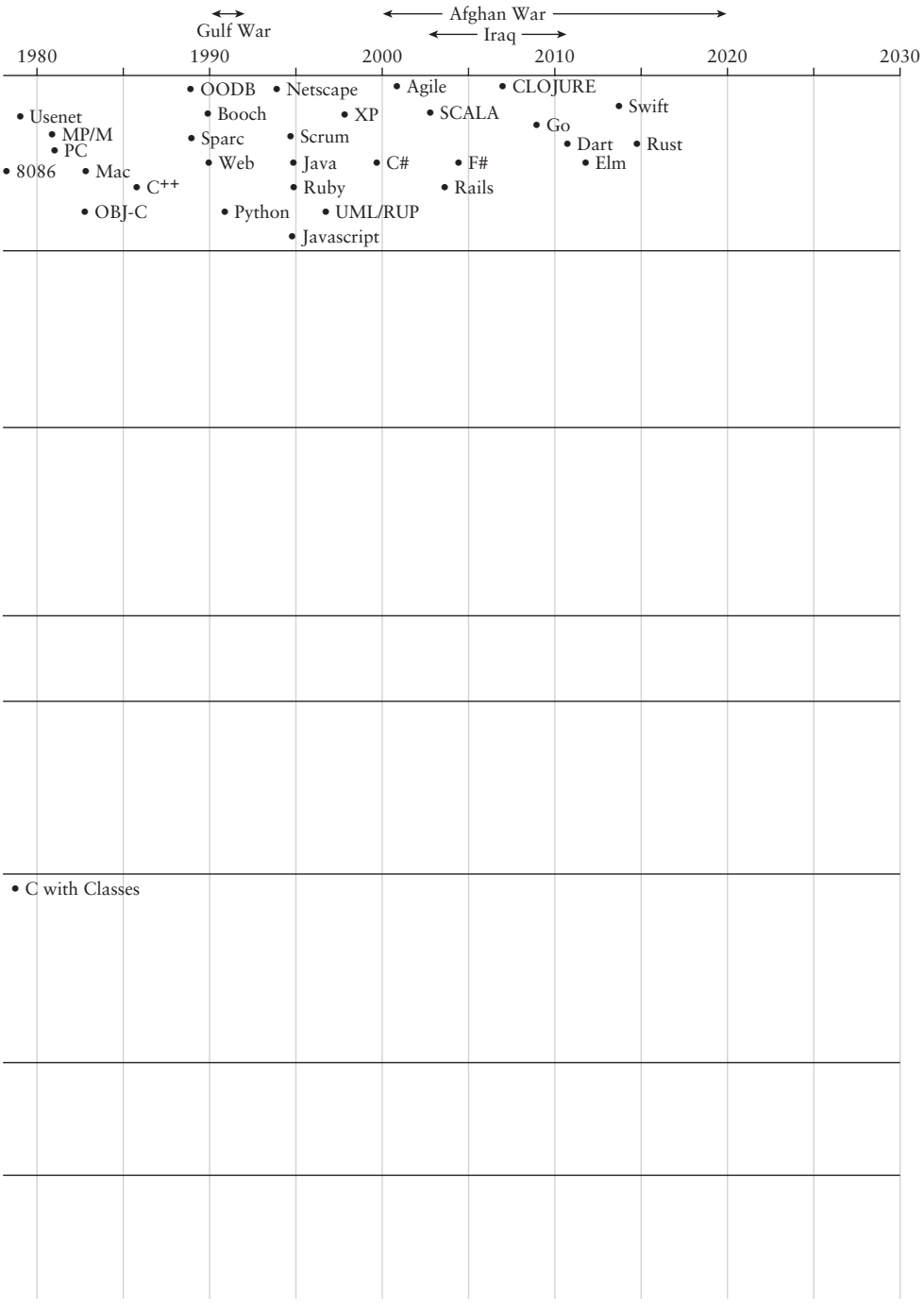
9/11 and the dotcom bust slowed us down for a few years, but then came the Ruby/Rails revolution, and then the mobile revolution. And then the internet was *everywhere*. The social networks blossomed and then decayed while AI reared up to threaten everything.

And that brings us to now and to thoughts of the future. All of that, and more, is what we will be talking about in the pages of this book. So, if you are ready, buckle up—because the ride is going to be a wild one.

Timeline

The people and events described through the stories in this book are depicted upon this timeline. As you read the stories and encounter the events, you can place them in context by finding them here. For example, you may find it interesting to know that FORTRAN and Sputnik are coincident in time, or that Ken Thompson joined Bell Labs before Dijkstra said that GOTO statements were harmful.


WW2
Korean War
Vietnam War
1930
1940
1950
1960
1970
Events
• Pearl Harbor
• IBM 701
• IBM 7090
• IBM 704
• UNIVAC 1107
VAX •
• Trinity
• UNIVAC 1108
• CP/M
• Manchester Baby
• PDP7
Apple][•
• Sputnik
• PDP8
• 8080
• PDP11
• ECP-18
• DN30&335
• H200
Hilbert
• Hilbert's Challenge
• Hilbert Dies
• Von Neumann Dies
• Gödel's Incompleteness
• Hilbert: No Math in Gottingen
• Turing Decideability
• Turing Dies
• Von Neumann Meets Turing
• Von Neumann to London (NCR)
Turing to Bletchley Park •
• Von Neumann to Los Alamos
• Von Neumann sees Mark I & ENIAC. Writes EDVAC draft.
Hopper
• Hopper Joins Mark I
• Betty Snyder's Merge Sort
• Mark I Symposium
• Hopper Joins UNIVAC
• Hopper: Automatic Programming
• A-O
• Symposium: Automatic Programming
• B-O Flowmatic
• CODASYL
• COBOL
Backus
• SSEC
• Speedcoding
• Fortran
• BNF ALGOL
Dijkstra
• Cambridge EDSAC
• THE
• MCC ARRA
• Gatlinburg
• Programmer
• Go to Harmful
• FERTA
• ARMAC
• Min Path
• XI
• First ALGOL
• Dark Future
Nygaard
• Nygaard NDRE
• Dahl NDRE
• Monte Carlo Compiler
• Simula Spec
• Nygaard pitches UNIVAC
• 1107 Delivered
• Simula I
• Simula 67
Simula 67 Commercialized •
Stroustrup Arrhus •
Kemeny
• Kemeny Hears Von Neumann on EDVAC
• Hired at Dartmouth
• LGP-30
• DN30 & DN235
• BASIC & Timesharing
Ritchie
• Ken Thompson Bell
• Dennis Ritchie Bell
• Dennis Ritchie No Thesis
• Brian Kernighan Bell
• MULTICS Killed
• UNIX PDP7
• UNIX PDP11
• C
K&R •


Gulf War
Afghan War
Iraq
1980
1990
2000
2010
2020
2030
• Usenet
• MP/M
• PC
• 8086
• Mac
• C++
• OBJ-C
• OODB
• Booch
• Sparc
• Web
• Python
• Netscape
• XP
• Scrum
• Java
• Ruby
• Javascript
• Agile
• C#
• UML/RUP
• SCALA
• F#
• Rails
• CLOJURE
• Go
• Swift
• Dart
• Rust
• Elm
• C with Classes

This page intentionally left blank

About This Book

Before we get started, there are a few points about the book, and its author, that I think you should be aware of.

- At the time of this writing, I have been a programmer for 60 years, although perhaps the years from ages 12 to 18 should be counted differently. Still, from 1964 until today, I have participated in the vast majority of the "computer age." I have seen and lived many of the important and even foundational events within this field. And so what you are about to read was written by someone who is part of a small and shrinking group of early travelers in this field. And though that group cannot claim to be among the earliest of travelers, we *can* claim to have accepted the baton from their hands.
- This work spans two centuries of time. Many may find the names and ideas mentioned in the telling of these stories to be unfamiliar—lost in the shadows of time. Therefore, at the end of this work, there is a *Glossary of Terms* and a *Cast of Supporting Characters*.
- The *Glossary of Terms* contains a description of most of the hardware mentioned in the text. If you see a computer or a device that you don't recognize, see if you can find it there.

- The *Cast of Supporting Characters* is a list of the people mentioned in passing in the pages that follow. The list is quite long, and yet it is far too short. It names just some of the people who had either a direct or indirect influence on the industry of computer programming. Some of the folks mentioned in the book have been lost in the mist of time and in the fog of internet search engines. Look through those names and be amazed at who you find there. Look again and realize that the list just barely scratches the surface. Look yet again at the dates when these people passed. Most of them passed very recently indeed.

Acknowledgments

Yet again, I offer my thanks to the folks at Pearson who have worked so hard to publish this book: Julie Phifer, Harry Misthos, Julie Nahil, Menka Mehta, and Sandra Schroeder. And thanks also to the production team who polished the content: Maureen Forys, Audrey Doyle, Chris Cleveland, and others. Working with them is always a pleasure.

Thank you to Andy Koenig and Brian Kernighan for helping me to make connections.

A special thanks to Bill and John Ritchie for providing me with so many wonderful insights into their brother "Dear old DMR".

Thanks to Michael Paulson (aka, The name is ThePrimeagen) for the lovely Foreword.

Thanks to Tom Gilb for your hospitality, your insight, and for one of the most entertaining Afterwords I have ever read.

Thanks to Grady Booch, Martin Fowler, Tim Ottinger, Jeff Langr, Tracy Brown, John Kern, Mark Seeman, and Heather Kanser for reviewing

the manuscript when it was in much rougher shape. Their help made it much better.

As always, I thank my wonderful and beautiful wife—the love of my life—and my four spectacular children and my ten equally spectacular grandchildren. They are my life. Writing about software is just for fun.

Lastly, I must offer thanks that my life is perfect—I live in paradise.

About the Author

Robert C. Martin (Uncle Bob) has been a programmer since 1970. He is founder of Uncle Bob Consulting, LLC, and cofounder with his son Micah Martin of Clean Coders, LLC. Martin has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He has authored and edited many books, including *Designing Object-Oriented C++ Applications Using the Booch Method*; *Pattern Languages of Program Design 3*; *More C++ Gems*; *Extreme Programming in Practice*; *Agile Software Development: Principles, Patterns, and Practices*; *UML for Java Programmers*; *Clean Code*; *The Clean Coder*; and *Functional Design: Principles, Patterns, and Practices*. A leader in the industry of software development, Martin served for three years as editor-in-chief of *The C++ Report*, and he served as the first chairman of the Agile Alliance.

This page intentionally left blank

6 Edsger Dijkstra: The First Computer Scientist

Edsger Dijkstra is one of the most famous names in software. He is the father of structured programming and the guy who told us not to use GOTO. He invented the semaphore construct and was the author of many useful algorithms. He cowrote the first working version of the ALGOL 60 compiler and participated in the design of early multiprogramming operating systems.

But these accomplishments are mere mile markers along a battle-torn path. The true legacy of Edsger Dijkstra is the ascendance of the abstract over the physical, a battle he fought with himself, with his peers, and with the profession at large, and eventually won.

He was a pioneer and a legend and a programmer, and perhaps not particularly humble. Alan Kay once affectionately said of him: "Arrogance in computer science is measured in nano-Dijkstras." He went on to explain that "Dijkstra was much more funny than annoying for anyone who had any sense of self." In any case, the story of Edsger Dijkstra is fascinating.

The Man

Edsger Wybe Dijkstra was born in Rotterdam on May 11, 1930. His father was a chemistry teacher and president of the Dutch Chemical

Society. His mother was a mathematician. So the boy was surrounded by math and technology.

During the German occupation of the Netherlands, his parents sent him away to the countryside until things calmed down. After the war, a lot was broken, but there was also a lot of hope.

Dijkstra graduated high school in 1948 with the highest possible marks in math and science.

Like all young people, I suppose, he rejected his parents' technical calling at first, and thought he'd be better suited to study law and represent the Netherlands at the UN. But the technical bug caught him at Leiden University, and his interests quickly changed to theoretical physics.

Those interests changed again in 1951 when his father arranged for him to go to Cambridge, in the UK, to attend a three-week course introducing programming on the EDSAC. His father thought that learning about the new tools of the day would enhance his career in theoretical physics. But the result was quite different.

Although he struggled with English, he very much enjoyed the class; and he learned a great deal. He said that those three weeks changed his life. While there, he met the director of the Mathematical Centre (MC) in Amsterdam, Adriaan van Wijngaarden, who offered him a part-time job as a programmer.

Dijkstra accepted that job in March of 1952. He was the first programmer in the Netherlands. He was hooked by how unforgiving computers were.

The MC did not have a computer at the time. They were trying to build the ARRA, an electromechanical machine. He helped design the instruction set and wrote programs that would have to wait until the machine was built.

It was here that he met Gerrit Blaauw, who had studied under Howard Aiken at Harvard in '52 and would go on to help design the IBM 360

with Fred Brooks and Gene Amdahl. Blaauw imported both material and many decent techniques into the MC from Aiken's laboratory.

Dijkstra and Blaauw worked together on the ARRA[1] and other projects. Of Blaauw, Dijkstra once told the following story:[2]

> "He was a very punctual man, and that was a good thing because for a couple of months [he] and I debugged the FERTA[3] that was installed at Schiphol Aircraft.[4] The place was a little hard to reach and, thank goodness, he had a car. So early in the morning, at 7:00, I would jump on my bicycle, go to the highway, hide my bicycle, and wait on the side until Gerrit came and picked me up. It was a grim winter, and it could only be done thanks to an old Canadian army code and due to the fact that, as I said, Gerrit was a very punctual man. I remember one horrible occasion that we had been debugging the FERTA all day long and it was 10:00 in the evening and his [car] was the last car in the […] parking lot, and it was beginning to snow, and the thing wouldn't start. I don't exactly remember how we got home, my guess is that Gerrit Blaauw fixed […] another bug that day."

Dijkstra loved the challenge of "being ingenious and being accurate." But the deeper he got into the efforts of the MC, the more he feared that programming was not an appropriate field for him. He was, after all, being groomed as a first-class theoretical physicist, and programming computers was not first-class anything at the time.

This personal dilemma came to a head in 1955 when he expressed his fears to Wijngaarden. Dijkstra said:

> "When I left his office a number of hours later, I was another man. [Wijngaarden explained] quietly that automatic computers were here to stay, that

1. An early electromechanical computer.

2. "A Programmer's Early Memories" by Edsger Dijkstra.

3. An early electronic computer derived from the ARRA.

4. An informal reference to Royal Dutch Aircraft Factory Fokker (in English). Also known as the Nederlandse Vliegtuigenfabriek (Dutch Aircraft Factory) in the Dutch area of Schiphol.

> we were just at the beginning and [I could] be one of the persons called to make programming a respectable discipline in the years to come. This was a turning point in my life…."[5]

In 1957 Dijkstra married Maria (Ria) Debets, one of the girls working as a "computer" in the MC. The officials in Amsterdam refused to recognize the profession of "programmer" that Dijkstra had written upon his marriage license. Of this, he says:

> "Believe it or not, but under the heading of 'profession' my marriage act shows the ridiculous entry 'theoretical physicist'!"

THE ARRA: 1952–1955

Wijngaarden hired Dijkstra to program the MC's new ARRA computer. The problem was that the ARRA didn't work. Five years before, Wijngaarden had traveled to Harvard and had seen Aiken's Mark I and its successors. Impressed, Wijngaarden hired some young engineers to build a similar but much smaller machine: the ARRA.

The ARRA was designed to be a stored-program *electromechanical* beast. It had a drum memory, 1,200 relays, and some vacuum tube flip-flops for registers. The relays had to be cleaned regularly, but the contacts degraded so rapidly that the switching time was undependable, making the machine unreliable. After four years of development and constant repair, the poor machine was apparently only good for generating random numbers.

That was not a joke. They had written a demonstration program that simulated the rolling of a 13-dimensional cubic die.[6] At one point, a minister of government came in to see this demo. They fearfully turned the machine on and, for once, it started printing proper random numbers. It worked!—for a moment. Then it unexpectedly halted. Wijngaarden,

5. "The Humble Programmer." *Communications of the ACM* 15, no.10 (Oct. 1972), 859–866.

6. Only a bunch of mathematical geeks could come up with *that* as a demo.

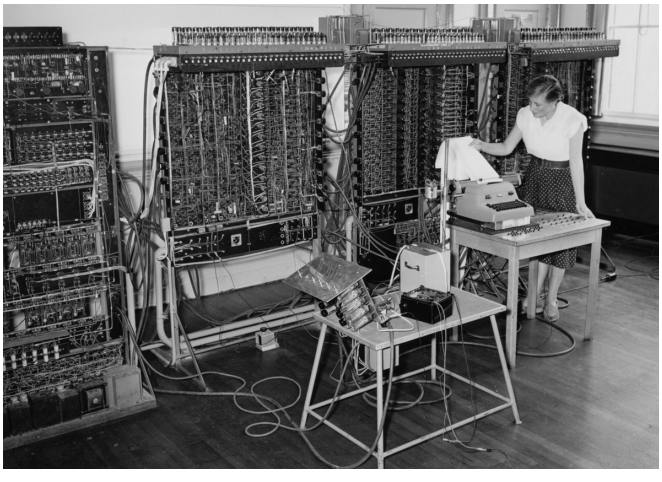
thinking on his feet, explained to the minister: "This is a highly remarkable situation: The cube is balancing on one of its corners and does not know which way to fall. If you push this button, you will give the cube a little push and the ARRA will continue its computation." The minister pushed the button, and the machine once again started printing random numbers. The minister replied: "That is very interesting," and left shortly thereafter.

The machine never worked again.

A frustrated Wijngaarden recruited Gerrit Blaauw, who had just gotten his Ph.D. at Harvard while working with Aiken on the Harvard Mark IV. Blaauw joined the team and in very short order convinced them that their machine had no hope of success, and that they needed to start from scratch with electronics and not electromechanical relays.

So the team started in on the design of the ARRA. Well, actually, it was the ARRA 2, but they didn't want to advertise that there had ever been an ARRA 1.

So they just called the new machine the ARRA.



Being the "programmer," Dijkstra would propose an instruction set to the hardware guys. They, in turn, would assess whether that instruction set

was practical to build. They'd make amendments and toss them back to Dijkstra. This iteration continued until they were all ready to "sign in blood." And then the hardware guys built the hardware, and Dijkstra started working on the programming manual and writing the IO primitives.

This division of labor established the idea that the hardware was a black box that executed the software, and the software was independent of the design of the hardware. This was an important step in computer architecture and a very early example of dependency inversion. The hardware depended on the needs of the software, rather than the software being subordinate to the dictates of the hardware.

They managed to complete the new machine 13 months later, in 1953. What's more, it worked—and they put it to work 24 hours a day calculating things like wind patterns, water movements, and the behavior of airplane wings. The machine completely replaced the team of women "computers" who had previously been doing these calculations on desk calculators. Those women became the programmers of the new machine.

At some point, they hooked a loudspeaker up to one of the registers and could listen to the rhythmic sounds as it operated. This was a good debugging tool because you could *hear* when it was executing properly and when it got caught in a loop.

The ARRA was designed as a binary machine at a time when most computers were using decimal (BCD). The electronics were vacuum tubes. It had two working registers, had 1,024 30-bit words of drum memory, and could execute ~40 instructions per second. The drum spun at 50 revs per second. The primary input device was five-channel paper tape. Output was typically sent to an automatic typewriter.

The instruction set that Dijkstra designed for the ARRA 2 should provide some insight into what he and his contemporaries thought a computer should be able to do: It was quite sparse, and focused much more on arithmetic than data manipulation. Later computers would reverse that decision!

Instructions in the ARRA 2 were 15 bits wide, so two instructions would fit into a 30-bit word. Those two instructions were referred to as *a* and *b*. So the machine could hold 2,048 instructions. The first five bits in an instruction were the operation code, and the last ten were the memory address (or an immediate value).

The two registers were named A and S. They were independently manipulated, but could also be used as a double-precision 60-bit number.

There were 24 instructions. Notice the lack of IO and indirect addressing. Notice also that Dijkstra used decimal to denote the operation codes. At the time, there was no thought of octal or hexadecimal:

```
0/n replace (A) with (A)+(n)
1/n replace (A) with (A)-(n)
2/n replace (A) with (n)
3/n replace (A) with -(n)
4/n replace (n) with (A)
5/n replace (n) by -(A)
6/n conditional control move to na
7/n control move to after na
8/n replace (S) with (s)+(n)
9/n replace (S) with (S)-(n)
10/n replace (S) with (n)
11/n replace (S) with -(n)
12/n replace (n) with (S)
13/n replace (n) with -(S)
14/n conditional control move to nb
15/n control move to nb
16/n replace [AS] with [n].[s]+[A]
17/n replace [AS] with -[n].[S]+[A]
18/n replace [AS] with [n].[S]
19/n replace [AS] with - [n].[s]
20/n divide [AS] by [n]:, place quotient in S, remainder in A
21/n divide [AS] by -[n]:, place quotient in S, remainder in A
22/n shift A->S, i.e. replace [AS] with [A].2^(29-n)
23/n shift S->A, i.e. replace [SA] with [S].2^(30-n)
24/n communication assignment
```

The two conditional control moves (jumps) executed only if the result of the last arithmetic operation was positive. The ARRA was a one's complement machine, which means there were two representations of zero, one positive and the other negative. So testing for zero by subtracting was perilous.

Notice that if you want to jump to instruction *a* or *b*, you have to use a *different* jump instruction. That must have been hellish to deal with when you inserted an instruction into your program.

The drum memory was arranged into 64 channels containing 16 words each. I infer that each channel corresponded to a track on the drum. I also infer that there were 64 read heads, and that each read head was routed to the electronics through relays. Those relays took from 20 ms to 40 ms to open or close. So the efficiency-minded programmer had to take care not to jump around between tracks.

What's more, it appears that the electronics for reading and writing the drum were able to read from two tracks at a time: one for instructions and another for data. This allowed the programmer to aggregate instructions on one track and data on another without having to change the state of the relays.

Just to add one extra complication, it appears that the actual tracks on the drum could only hold eight words around the drum circumference, but that each read/write head was actually a dual head that could read and write one or the other of two adjacent tracks. Thus, reading an entire 16-word channel required at least two revolutions of the drum.

Imagine having all that to think about while you are trying to write an interesting mathematical program.

Notice that the instruction set does not allow for indirect addressing. Either Dijkstra had not yet realized the importance of pointers, or the hardware designers could not easily implement indirection. So any indirect or indexed access had to be accomplished by modifying the addresses within existing instructions.

This was also the strategy for subroutines. The instruction set had no "call" instruction, nor any way to remember return addresses. So it was the programmers' job to store the appropriate jump instruction in the last instruction of a subroutine, before jumping to that subroutine.

The primitive nature of these machines and instruction sets likely had a dual effect upon Dijkstra. First, being so swamped by all the hideous detail and the deeply constrained environment likely precluded any serious consideration of computer science. But second, it also likely instilled within him the strong desire to shed all such detail and drive toward abstraction. As we shall see, that's exactly the path that Dijkstra took.

Indeed, there are hints of this battle between detail and abstract thinking in Dijkstra's "Functionele Beschrijving Van De Arra" ("Functional Description of the ARRA") from 1953. In that write-up, Dijkstra goes to great lengths to justify and explain the benefits of callable subroutines as opposed to the kind of "open subroutines" (code duplication) that Hopper had been using in the Mark I.

In his description of how to do a subroutine call, he says that "it would be nice to have control run through the same series of commands" instead of duplicating those commands over and over within the program, which "is quite bad [and] is wasteful of memory space." But then he turns around and complains that jumping to a subroutine in that fashion will cause a great deal of drum-switching time, adding a full revolution of the drum to each call and each return.

Coming to terms with this dichotomy is the story, and the legacy, of Edsger Dijkstra.

THE ARMAC: 1955–1958

The MC continued to make improvements to the design, and they built new machines based upon the ARRA.

The FERTA was built two years later (1955) and was installed at the Fokker factory next to Schiphol Airport. It was used to calculate matrices used for the design of the wings of the F27 Friendship. The architecture was similar to the ARRA, but memory was expanded to 4,096 34-bit words and speed was doubled to ~100 instructions per second, probably because the density of each track on the drum had doubled.

The ARMAC was another ARRA derivative that followed one year later (1956). It could execute 1,000 instructions per second because a small bank of core memory was used to buffer tracks on the drum. This machine had 1,200 tubes and consumed 10 kilowatts.

This was a period of intense hardware experimentation and improvement, but not much improvement in the software architecture—particularly the instruction set. However, the increase in memory and speed allowed Dijkstra to consider a previously untenable problem.

DIJKSTRA'S ALGORITHM: THE MINIMUM PATH

"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning [in 1956] I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame."

—Edsger Dijkstra, 2001

Dijkstra chose the problem because he wanted to demonstrate the power of the ARMAC with a non-numerical problem that normal people could understand. He wrote the demonstration code to find the shortest path from one city to another through a simplified traffic network between 64[7] cities in the Netherlands.

The algorithm is not a difficult one to understand. It's a matter of a few nested loops, a bit of sorting, and a fair bit of data manipulation. But imagine writing this code without pointers. Without recursion. Without a call instruction for your subroutines. Imagine having to access your data by modifying the addresses within individual instructions. Further, consider the problems of optimizing this program to keep track of data from being flushed in and off the drum from the core buffer. My mind rebels at the thought!

Having dispensed with the demonstration, Dijkstra then created a similar algorithm to solve a more practical problem. The backplane of the next computer, the X1, had a vast number of interconnections, and copper is a precious metal. So he wrote the minimum spanning tree algorithm to minimize the amount of copper wire used on that backplane.

Solving these kinds of graph algorithms was a step toward the computer science that we recognize today. Dijkstra used the computer to solve problems that were not strictly numerical in nature.

ALGOL AND THE XI: 1958–1962

Dijkstra's efforts at the MC were taking place at a formative time in computing. Hopper's 1954 symposium on Automatic Programming[8] had generated a lot of buzz. At that symposium, J. H. Brown[9] and J. W.

7. Yes, he represented each city with a 6-bit integer.

8. See Compilers: 1951–1952, in Chapter 4.

9. From Willow Run Research Center.

Carr III[10] presented a paper on their efforts to create a universal programming language that was machine independent. Saul Gorn[11] presented a similar paper. Both papers advocated for divorcing the structure of the language from the physical structure of the hardware.

This was a revolutionary concept that flew in the face of the thinking of the day. Languages like FORTRAN, MATH-MATIC, and eventually even COBOL were strongly wed to the physical architecture of the machines of the day. This strong binding was thought to be necessary to achieve efficiency of execution. Brown and Carr acknowledged that universal languages might be slower, but they argued that they would reduce programming time and programmer error.

As an example of the wedding between language and physical architecture, consider the type `int` in the C language. Depending upon the machine, it is either a 16-, 32-, or 64-bit integer. I once worked on a machine in which it was an 18-bit integer. In C, an `int` is wedded to the machine word. Many languages of the day simply assumed that the data format of the language was the data format of the machine, including memory layout. Subroutine arguments were assigned fixed locations, precluding any kind of recursion or reentrancy.

In May of 1958, Carr and several others met in Zurich to begin the definition of a universal machine-independent programming language. They named it ALGOL.[12] The stated goal was to create a language based upon mathematical notation that could be used for the publishing of algorithms and mechanically translated into machine programs.

John Backus created a formal notation to describe an early (1958) version of the language. Peter Naur recognized the genius of the notation

10. John Weber Carr III (1923–1997).

11. From Ballistics Research Lab.

12. International ALGOrithmic Language.

and named it Backus Normal Form (BNF[13]). He then used it to specify the 1960 version of ALGOL.

Meanwhile, Dijkstra and the MC were busy building the X1 computer. This was a fully transistorized machine, with up to 32K of 27-bit words. The first 8K of the address space was read-only memory and contained boot programs and a primitive assembler. The X1 was one of the first machines to have a hardware interrupt—something that Dijkstra would make strategic use of. The machine also had an index register! At last, real pointers! Otherwise, it was similar in many ways to the ARRA and ARMAC machines.

The memory cycle time was 32 μs and the add time was 64 μs. It could therefore execute over 10,000 instructions per second.

In 1959 Wijngaarden and Dijkstra joined in the effort to define the ALGOL language. In 1960, with the BNF specification complete, Dijkstra and his close colleague Jaap Zonneveld surprised the computing world by quickly writing a compiler for the X1.

13. In 1964 Donald Knuth recommended that the name be changed to Backus–Naur Form.

"Surprised" might be an understatement. Naur wrote: "The first news of the success of the Dutch project, in June 1960, fell like a bomb in our group."[14] That bomb might explain some of the tension that arose later. I imagine Dijkstra & Co were viewed as upstarts who scooped everybody else.

Of their success, Dijkstra wrote:[15] "The combination of no prior experience in compiler writing and a new machine [the X1] without established ways of use greatly assisted us in approaching the problem of implementing ALGOL 60 with a fresh mind."

One of the techniques Dijkstra and Zonneveld used was "dual programming." Each would independently implement a feature and then they would compare the results. Dijkstra called this an "engineering approach." It has been suggested[16] that this approach so significantly reduced their debugging time that it led to a net reduction in their development time.

This was at a time when most teams implementing the language were debating which features to leave in or out. Dijkstra and Zonneveld left nothing out. They implemented the full specification—*in six weeks*.

Indeed, they even implemented the part of the specification that virtually everyone had planned on omitting: recursion.

This is ironic because recursion had been *voted out* of the language. Dijkstra and John McCarthy had strenuously advocated for recursion, but the rest of the committee thought that recursion was too inefficient and was otherwise useless. However, the wording of the resolution was ambiguous enough that Dijkstra sneaked the feature in anyway.

14. Daylight, p. 46.

15. Daylight, p. 57.

16. Private correspondence with Tom Gilb.

This got him into some trouble with the other members of the ALGOL team. Indeed, the issue of recursion versus efficiency generated quite a bit of tension. In a meeting in 1962, one of the members[17] got up and directed a nasty comment toward Dijkstra, Naur, and other proponents of recursion[18]—a comment that was greeted with loud applause and laughter:

> "And the question is—to state it once more—that we want to work with this language, really to work and not to play with it, and I hope we don't become a kind of ALGOL play-boys."

One reason that Dijkstra and Zonneveld were successful when so many other teams were still struggling was because Dijkstra created an abstraction boundary between the language and the machine. The compiler converted ALGOL source code into a p-code,[19] and an associated runtime system interpreted that p-code. Nowadays, we would call that runtime system a VM.[20]

This division allowed the language compiler to ignore the machine itself—which was one of the goals originally sought by Carr, Brown, Gorn, and the ALGOL team. Ignoring the machine made the compiler much easier to write. In short, Dijkstra invented a machine that was perfect for ALGOL, and then he made up the difference in his runtime system.

Of course, this caused certain inefficiencies. Simulating a p-code is always slower[21] than raw machine language. But Dijkstra justified this by stalwartly refusing to worry about efficiency in the short term. His focus

17. Gerhard Seegmüller.

18. Daylight, p. 44.

19. Portable code. A typically numeric code representing the instructions of a virtual machine.

20. A virtual machine like the JVM or the CLR.

21. Especially since just-in-time compilers were decades in the future.

was more upon the long-term direction of language and computer architecture. Thus, he said:[22]

> "In order to get as clear a picture as possible of the real needs of the programmer, I intend to pay, for a while, no attention to the well-known criteria 'space and time'. "

He went on to say:

> "I am convinced that [the] problems [of program correctness] will prove to be much more urgent than, for example, the exhaustive exploitation of specific machine features…."

And just to cap things off:

> ". . .Recursion] is such a neat and elegant concept that I can hardly imagine that it will not have a marked influence on the design of new machines in the near future."

And, of course, he was right. The PDP-11, with all its lovely index registers, one of which was a dedicated stack pointer, was only a decade away.

At the very start of the ALGOL project, Dijkstra and Zonneveld were concerned that Wijngaarden would (as was his wont) take undue credit. So the two clean-shaven programmers conspired to remain unshaven until the compiler worked. Only those with a beard could take credit for the compiler. Six weeks later, with the compiler working, Zonneveld shaved, but Dijkstra remained bearded from that point on.

THE GATHERING GLOOM: 1962

With the success of ALGOL 60 under his belt, Dijkstra looked to the future in a 1962 paper titled "Some Meditations on Advanced

22. Daylight, p. 59.

Programming." His first assertion is that software is an art that must become a science:

> "Therefore I would like to draw your attention in particular to those efforts and considerations which try to improve 'the state of the Art' of programming, maybe to such an extent that at some time in the future we may speak of 'the state of the Science of Programming'."

He then goes on to paint a rather gloomy picture of "the state of the art." He says that this change from an art to a science is "very urgent" because:

> ". . . The programmer's world is a very dark one with only just the first patches of a brighter sky appearing at the horizon."

He explains this by describing what it had, up to then, been like to be a programmer given "nearly impossible jobs" using machines that have capabilities "exhausted to slightly beyond their utmost limits." He explains that programmers under such circumstances fall back on "curious and tricky ways" to cajole their systems into working.

He calls the discipline of programming "extremely crude and primitive" and "unhygienic," and complains that the creativity and shrewdness of the programmers encourage the hardware designers to "include all kinds of curious facilities of doubtful usability." In other words, bad machines make bad programmers who encourage even worse machines.

What machines was he railing against? Rumor has it that it was the machines made by IBM.

As you read on in this paper, you realize that his specific complaint is that the machine architecture of the day makes recursion difficult and inefficient, and this tempts the compiler writers to constrain the language to prevent recursion.

He concludes the piece with an appeal to "elegance" and "beauty" and claims that unless programmers are given languages that are "charming"

and "worthy or our love," they will likely not create systems of "superior quality."

This paper came at a time when the concerns of the industry were focused on quite a different problem. Those were the early days of "the software crisis." It was becoming ever more obvious that projects, if they were delivered at all, were breaking budgets and schedules; were inefficient, buggy, and did not meet requirements; and were unmanageable and unmaintainable.

I presume that sounds familiar to you. The software crisis never really ended. We've just kind of learned to accept and live with it.

Dijkstra's solution was science, elegance, and beauty. And as we shall see, he wasn't wrong.

THE RISE OF SCIENCE: 1963–1967

With a working ALGOL compiler, and thereby freed from the confines of the physical hardware, Dijkstra was able to experiment with all manner of computer science issues. One of those early issues was multiprogramming. Dijkstra captained a team of five other researchers at the Eindhoven University of Technology who embarked on a project to build a multiprogramming system named the THE[23] Multiprogramming System.

Dijkstra described this system in a famous paper[24] that he submitted to the *Communications of the ACM* in 1968. The structure of that system was unique at the time, and showed just how far Dijkstra had come from his ARRA days.

This system ran on the X8 computer, which was a much faster and more capable derivative of the X1. The X8 had a minimum of 16K 27-bit

23. Technische Hogeschool Eindhoven.

24. "The Structure of the "THE"-Multiprogramming System." See this chapter's References section.

words, expandable to 256K. It had a 2.5 μs core memory cycle time, allowing it to execute *hundreds of thousands* of instructions per second. It had indirect addressing and hardware interrupts[25] that included IO and a real-time clock. For the day, it was an ideal platform for experimenting with multiprocessing.

Science

The architecture of the THE system was quite modern. Dijkstra was very careful to separate it into black box layers. He did not want high-level layers knowing about the intricacies of the layers at lower levels.

At the lowest layer (Layer 0) was processor allocation—this layer decided which processor would be allocated to a particular process. Above this layer, the executing program had no idea which processor it was running on.

The next layer up (Layer 1) was memory control. The system had a primitive virtual memory capability. Pages in the core could be swapped to and from the drum. Above this layer, the program simply assumes that all program and data elements are accessible in memory.

The next level (Layer 2) controlled the system console.[26] It was responsible for ensuring that individual processes could communicate with their users with the keyboard and printer. Above this level, each process simply assumed that it had undivided access to the console.

Layer 3 controlled IO and the buffering of IO devices. Above this level, user programs simply assumed that data going in and out of various devices was a continuous stream requiring no management.

25. He called it "an interrupt system to fall in love with." I imagine that's the last time anyone said that about interrupts.

26. It may seem strange that an entire layer was dedicated to a single IO device. But the system console was the only device that was shared among all the processes, and it therefore deserved special attention. Nowadays, we would likely handle this differently, but back then a better solution wasn't clear. I remember working on systems at the time that faced the same issue.

Layer 4 was where user programs executed.

In the mid-1960s, the discipline required in order to design an operating system with a carefully layered structure was brand new. The idea of creating layers of abstraction that isolated high-level policy from low-level detail was revolutionary. This was computer science at its best.

SEMAPHORES

As part of this development, the team faced the issue of concurrent update and race conditions. To address this, Dijkstra came up with the semaphore abstraction that we have all come to know and love.

A semaphore is simply an integer upon which two operations[27] are allowed. The *P* operation blocks the calling process if the semaphore is less than or equal to zero; otherwise, it decrements the semaphore and continues. The *V* operation increments the semaphore, and if the result is positive, it simply continues on. Otherwise, one of the blocked processes is released before continuing on.

As part of his write-up on semaphores, Dijkstra invents the terms *critical section* and *indivisible action*. A critical section is any portion of the program that is manipulating a shared resource in a way that must not be interrupted by another. That is, it does not allow concurrent update. An indivisible action is an action that must fully complete before a hardware interrupt is allowed. In particular, the increment and decrement operations upon semaphores must be indivisible.

Of course, nowadays we programmers know these concepts well. They were taught to us in our earliest years of coding. But Dijkstra and his team had to derive these concepts from first principles and formalize them into the abstractions that we now take for granted.

27. Named *P* and *V* for historical reasons.

Structure

Another radically scientific concept adopted by Dijkstra was the notion that programs were structures of sequential procedures. Each such procedure had a single entry and a single exit. All adjacent procedures viewed it as a black box. Such procedures could be sequential in that one executed after the other, or they could be iterative in that a procedure executed many times in a loop.

This black box structure of sequential and iterative procedures allowed Dijkstra's team to test the individual procedures in isolation and to create reasoned *proofs* that the procedures were correct.

The testing enabled by this structured approach was so successful that he proudly asserted:

> "The only errors that showed up during testing were trivial coding errors (occurring with a density of one error per 500 instructions), each of them located within 10 minutes [of] (classical) inspection by the machine and each of them correspondingly easy to remedy."

Dijkstra was convinced that this kind of structured programming was an essential part of good system design and was responsible for the significant success of the THE system. To dissuade detractors who claimed that its success was due to its relatively small size, Dijkstra wrote:

> ". . . I should like to venture the opinion that the larger the project, the more essential the structuring!"

Proof

And this is where things start to get a little strange. In his write-up of the THE system, Dijkstra makes a remarkable claim:

> ". . . The resulting system is guaranteed to be flawless. When the system is delivered, we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation."

Dijkstra made this claim because he believed that he and his team had proven, *mathematically*, that the system was correct. This is a view that Dijkstra would hold, promote, and evangelize for the rest of his career. It is also the source of what I consider to be his greatest mistake. In his view, programming was mathematics.

Indeed, in *On the Reliability of Programs*,[28] Dijkstra asserts that "programming will become more and more an activity of mathematical nature." And in this, Edsger Wybe Dijkstra was, in my view, dead wrong.

MATHEMATICS: 1968

The idea of software as a form of mathematics is a seductive one. There are, after all, many adjacencies. The early programs written by Hopper and her crew at Harvard were all numerical, and deeply mathematical in nature. The use of the Harvard Mark I to compute the solutions of systems of partial differential equations describing the implosion of the plutonium core of the Fat Man bomb required a very deep mathematical ability indeed.

It was, after all, Turing who claimed that programmers were "mathematicians of ability." And it was Babbage who was driven by the need to ease the burden of the creation of mathematical tables.

So when Dijkstra wrote that software would become more and more a matter of mathematics, he was standing on pretty firm historical ground. He therefore focused his endeavors on mathematically *proving* software programs correct.

We can plainly see this in his *Notes on Structured Programming*,[29] as he shows us the basic mathematical mechanisms for proving a simple algorithm correct. He outlines those mechanisms as enumeration, induction, and abstraction.

28. Dahl, Dijkstra, and Hoare, p. 3.

29. Dahl, Dijkstra, and Hoare, p. 12.

He uses enumeration to prove that two or more program statements in sequence achieve their desired goals while maintaining stated invariants. He uses induction to prove the same things about a loop. He uses abstraction as his primary motivation for black box structure.

And then, using all three of those mechanisms, he presents a proof of the following algorithm for computing the integer remainder of `a/d`:

```
a ⩾ 0 and d > 0.
"integer r, dd;
r: = a; dd: = d;
while dd ⩽ r do dd: = 2*dd;
while dd ≠ d do
    begin dd: = dd/2;
          if dd ⩽ r do r: = r − dd
    end".
```

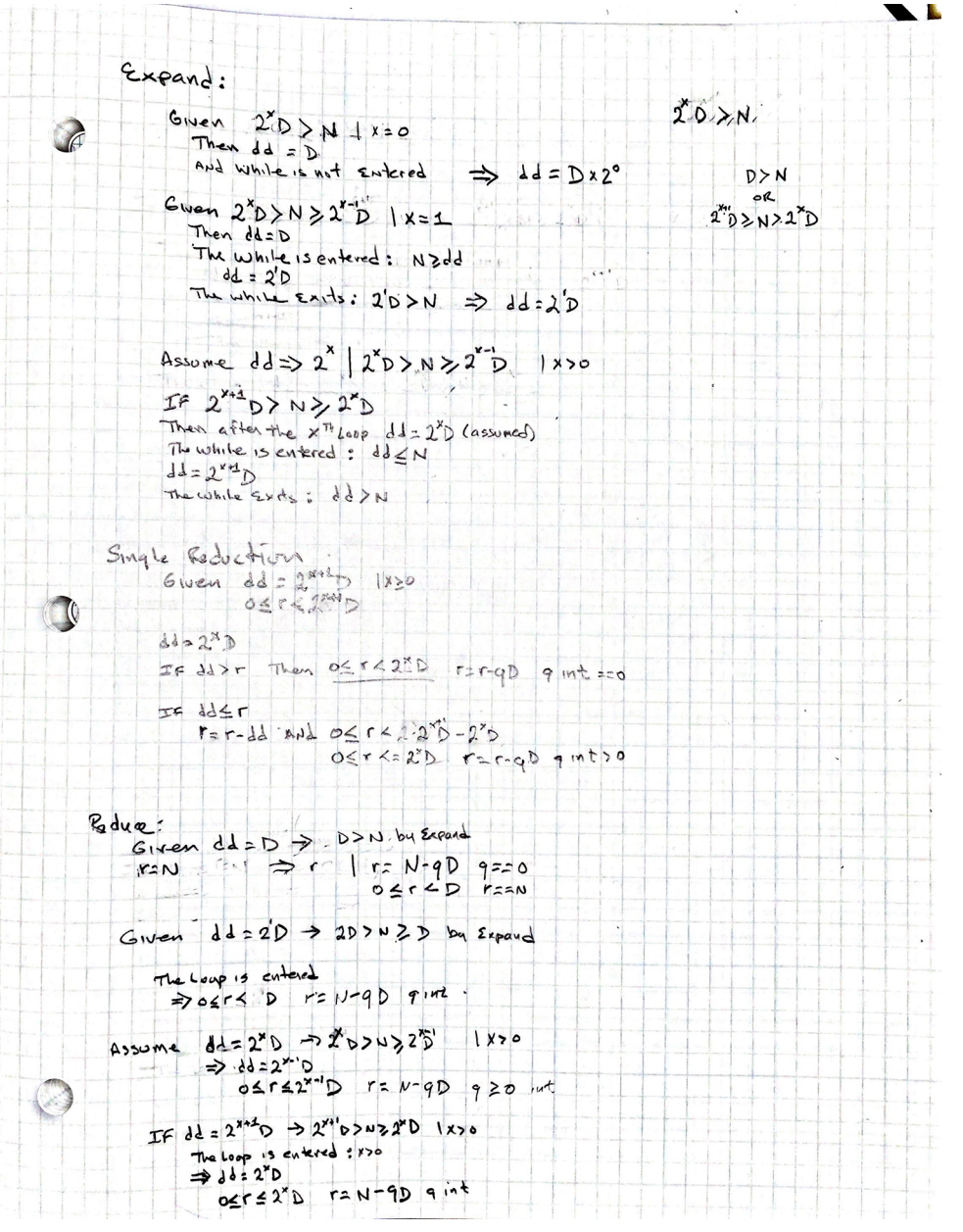
This is a lovely little algorithm that runs in logarithmic time using a shift-and-subtract approach. Any PDP-8 programmer would be proud.

The proof that Dijkstra presented was two pages long, followed by another page of notes. My own proof, based upon Dijkstra's, is a bit shorter but no less imposing (see next page).

It ought to be very clear that no programmer[30] would accept this as part of a viable process for writing software. As Dijkstra himself said:

> "The pomp and length of the above proof infuriate me […] I would not dare to suggest (at least at present!) that it is the programmer's duty to supply such a proof whenever he writes a simple […] program. If so, he could never write a program of any size at all!"

30. Other than, perhaps, those in an academic environment doing research on formal proofs of algorithms.



Expand:

Given $2^x D > N \mid x = 0$
Then $dd = D$
And while is not entered $\Rightarrow dd = D \times 2^0$

$2^x D \geq N$

$D > N$
OR
$2^{x-1} D \geq N > 2^x D$

Given $2^x D > N \geq 2^{x-1} D \mid x = 1$
Then $dd = D$
The while is entered: $N \geq dd$
$dd = 2^1 D$
The while exits: $2^1 D > N \Rightarrow dd = 2^1 D$

Assume $dd \Rightarrow 2^x \mid 2^x D > N \geq 2^{x-1} D \mid x > 0$

IF $2^{x+1} D > N \geq 2^x D$
Then after the x^{th} Loop $dd = 2^x D$ (assumed)
The while is entered: $dd \leq N$
$dd = 2^{x+1} D$
The while exits: $dd > N$

Single Reduction
Given $dd = 2^{x+1} D \mid x \geq 0$
$0 \leq r < 2^{x+1} D$

$dd = 2^x D$
IF $dd > r$ Then $0 \leq r < 2^x D$ $r = r - qD$ q int $= 0$

IF $dd \leq r$
$r = r - dd$ AND $0 \leq r < 2^{x+1} D - 2^x D$
$0 \leq r < 2^x D$ $r = r - qD$ q int > 0

Reduce:
Given $dd = D \rightarrow D > N$ by Expand
$r = N$ $\Rightarrow r \mid r = N - qD$ $q = 0$
$0 \leq r < D$ $r = N$

Given $dd = 2^1 D \rightarrow 2D > N \geq D$ by Expand

The Loop is entered
$\Rightarrow 0 \leq r < D$ $r = N - qD$ q int

Assume $dd = 2^x D \rightarrow 2^x D > N \geq 2^{x-1} D \mid x > 0$
$\Rightarrow dd = 2^{x-1} D$
$0 \leq r \leq 2^{x-1} D$ $r = N - qD$ $q \geq 0$ int

IF $dd = 2^{x+1} D \rightarrow 2^{x+1} D > N \geq 2^x D \mid x > 0$
The loop is entered: $x > 0$
$\Rightarrow dd = 2^x D$
$0 \leq r \leq 2^x D$ $r = N - qD$ q int

But then he goes on to say that he felt the same kind of "fury" when examining the early theorems of Euclid's plane geometry. And this is where Dijkstra's dream shows up.

Dijkstra dreamed that, one day, there would be a body of theorems and corollaries and lemmas that programmers could draw from. That they would not be relegated to proving their code in such horrid detail, but rather would restrict themselves to techniques that were already proven, and could create new proofs based upon the old ones without all the "pomp and length."

In short, he thought software was going to be like Euclid's *Elements*, a mathematical superstructure of theorems, a vast hierarchy of proofs, and that programmers would have little to do other than assemble their own proofs from the resources in that vast hierarchy.

But here I sit, in the waning months of 2023, and I do not see that hierarchy. There is no superstructure of theorems. The *Elements* of software has not been written. Nor do I believe it ever will.

There is an odd parallel between Hilbert and Dijkstra in this regard. Both were seeking a grand edifice of truth that simply cannot exist.

Dijkstra's dream has not been achieved. In all likelihood, it cannot be achieved. And the reason for that is that software is not a form of mathematics.

Mathematics is a positive discipline—we prove things correct using formal logic. Software, as it turns out, is a negative discipline—we prove things incorrect by observation. If that sounds familiar, it should. Software is a science.

In science, we cannot prove our theories correct; we can only observe them to be incorrect. We perform these observations with well-designed and well-controlled experiments. Likewise, in software, we almost never

try to prove our programs correct. Instead, we detect their incorrectness by conducting observations in the form of well-designed tests.

Testing was something that Dijkstra complained about by saying: "Testing shows the presence, not the absence, of bugs." And, of course, he was correct. What he missed, in my opinion, is that his statement proved that software is not mathematics, but rather a science.

STRUCTURED PROGRAMMING: 1968

In 1967, after all the successes he had enjoyed, the MC disbanded Dijkstra's group because they saw no future in computer science. Dijkstra, for that reason among others, descended into a six-month severe depression for which he was hospitalized.

Upon recovery, Dijkstra started to really stir things up.

In an ironic twist, it was Dijkstra's dream of mathematics that led him to what is perhaps his most valuable contribution to computer science and the software craft: structured programming. In hindsight, we appreciate that value, but at the time, it stirred up quite a controversy.

In 1967 Dijkstra gave a talk at the ACM Conference on Operating System Principles in Gatlinburg, TN. He had lunch with some of the attendees, and the discussion turned to Dijkstra's view on the GOTO statement. Dijkstra explained to them why GOTO introduced complexity into programs. The group was so impressed by this discussion that they encouraged him to write an article for *The Communications of the ACM* on the topic.

So Dijkstra wrote a short article describing his views. He titled it "A Case Against the Go To Statement." He submitted it the editor, Niklaus Wirth,[31] who was so excited to publish it that in March of 1968 he

31. Yeah, THAT Niklaus Wirth. You know: the inventor of Pascal.

rushed it through as a letter to the editor rather than a fully reviewed article. Wirth also changed the title to "Go To Statement Considered Harmful."

That title has echoed down through the decades.

The title also raised the ire of an entire cohort of programmers, who were horrified at the very idea. It's not clear to me whether those programmers actually read the article before declaring their angst. Whatever the case may be, the programming journals lit up like a Christmas tree.

I was a very young programmer at the time, and I remember quite well the fracas that ensued. There was no Facebook or Twitter (X) at the time, so the flame wars were conducted in letters to the editors all around the industry.

It took five to ten years or so for things to settle down. In the end, Dijkstra won that war. As a rule, we programmers do not use GOTO.

Dijkstra's Argument

In 1966 Corrado Böhm and Giuseppe Jacopini wrote a paper titled "Flow Diagrams, Turing Machines, and Language with Only Two Formation Rules." In this paper, they proved that "every Turing machine is reducible to, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration." In other words, every program can be reduced to statements in sequence or statements in loops. Period.

This was a remarkable finding. It was generally ignored because it was deeply academic. But Dijkstra took it very seriously, and the argument he laid down in his article is hard to refute.

In short, if you want to understand a program—if you want to prove the program correct—you need to be able to visualize and examine the execution of that program. That means you need to turn the program

into a sequence of events in time. These events need to be identifiable with some kind of label[32] that ties them back to the source code.

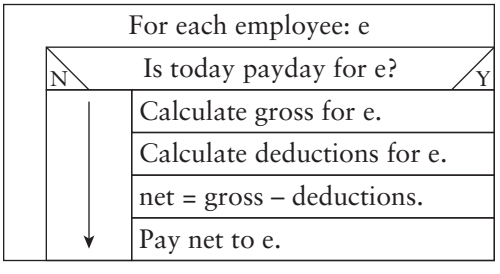
In the case of simple sequential statements, that label is nothing more than the line number of the source code statement. In the case of loops, that label is the line number adorned with the state of the loop; for example, the *n*th execution of line *l*. For function calls, it is the stack of line numbers that represent the calling sequence. But how do you construct a reasonable label if any statement can jump to any other statement using a GOTO?

Yes, it's possible to construct those labels, but they would have to consist of a long list of line numbers, each adorned with the state of the system.

Constructing proofs using such unmanageable labels is simply unreasonable. It's too much to ask.

So Dijkstra recommended maintaining provability by eliminating unconstrained GOTOs and replacing them with the three structures we've come to know and love: *sequence*, *selection*, and *iteration*. Each of those structures is a black box with a single entry and a single exit, and each is likely composed of the same structures in a recursive descent.

As a simple example, consider this payroll algorithm:



32. He used the term *coordinate*. Consider this alongside Babbage's notation for dynamics.

Nassi–Shneiderman diagrams like this constrain the algorithm to the three structures. You can see the *iteration* around the outside, the *selection* as the first element of the iteration, and the four *sequences* along the *Y* path of the selection. Each of those four sequences are subsequences that are themselves composed of sequences, selections, and iterations.

Why, if we are not going to follow Dijkstra's dream of creating proofs, is this strategy valuable? Because even though we don't write proofs for our code, we want our code to be *provable*. Provable code is code that we can analyze and reason about. Indeed, if we keep our functions small, simple, and provable, then we needn't even create Dijkstra's labels.

In any case, Dijkstra won the argument rather decisively; the GOTO statement has been all but completely driven out of our modern menagerie of languages.

It would be a mistake, however, to think that Dijkstra's view of structured programming was simply the absence of GOTO. That's the part that most of us remember, but his intention was far more involved than that. It went as far as layers of architecture and the directions of dependencies. But I'll leave you, gentle reader, to discover this for yourself within his wonderful writings.

References

Apt, Krzysztof R., and Tony Hoare (Eds.). 2002. *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. ACM.

Belgraver Thissen, W. P. C., W. J. Haffmans, M. M. H. P. van den Heuvel, and M. J. M. Roeloffzen. 2007. "Unsung Heroes in Dutch Computing History." https://web.archive.org/web/20131113022238/http://www-set.win.tue.nl/UnsungHeroes/home.html.

Computer History Museum. "A Programmer's Early Memories by Edsger W. Dijkstra." Lecture presented at the First International Research Conference on the History of Computing, Los Alamos, New Mexico, summer of 1976. Posted June 10, 2022. 24:23. Posted on YouTube on June 10, 2022. (Available at the time of writing.)

computingheritage. "Remembering ARRA: A Pioneer in Dutch Computing," 9:13. Posted on YouTube on June 4, 2015. (Available at the time of writing.)

Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming*. Academic Press.

Daylight, Edgar G. 2012. *The Dawn of Software Engineering: From Turing to Dijkstra*. Lonely Scholar Scientific Books.

Dijkstra, E. W. 1953. "Functionele Beschrijving Van De Arra." Mathematisch Centrum, Amsterdam. https://ir.cwi.nl/pub/9277.

Dijkstra, Edsger. 1968. "The Structure of the 'THE'-Multiprogramming System." *Communications of the ACM*. www.cs.utexas.edu/~EWD/ewd01xx/EWD196.PDF.

Dijkstra, Edsger W. "Edsger Dijkstra - Oral Interview for the Charles Babbage Institute - 2001," 2:09:32. Posted on YouTube on Jan. 3, 2023.

Markoff, John. 2002. "Edsger Dijkstra, 72, Physicist Who Shaped Computer Era." *New York Times*, August 10, 2002. www.nytimes.com/2002/08/10/us/edsger-dijkstra-72-physicist-who-shaped-computer-era.html.

University of Cambridge, Computer Laboratory. 1999. "EDSAC99, EDSAC 1 and After: A Compilation of Personal Reminiscences." www.cl.cam.ac.uk/events/EDSAC99/reminiscences.

Van den Hove, Gauthier. 2009. "Edsger Wybe Dijkstra, First Years in the Computing Science (1951-1968)." MsCS Thesis, Universitē de Namur. https://pure.unamur.be/ws/portalfiles/portal/36772985/2009_VanDenHoveG_memoire.pdf.

Van Emden, Maarten. n.d. "Dijkstra, Blaauw, and the Origin of Computer Architecture." *A Programmer's Place*. https://vanemden.wordpress.com/2014/06/14/dijkstra-blaauw-and-the-origin-of-computer-architecture.

Van Emden, Maarten. n.d. "I Remember Edsger Dijkstra (1930-2001)." *A Programmer's Place*. https://vanemden.wordpress.com/2008/05/06/i-remember-edsger-dijkstra-1930-2002.

Wikipedia. "Dijkstra's algorithm." https://en.wikipedia.org/wiki/Dijkstra's_algorithm.

Wikipedia. "Edsger Dijkstra." https://en.wikipedia.org/wiki/Edsger_W._Dijkstra.

Index

C