PRENTICE HALL

Robert C. Martin Series

The Clean Coder

A Code of Conduct for Professional Programmers

Foreword by Matthew Heusser, Software Process Naturalist

Robert C. Martin

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for *The Clean Coder*

"'Uncle Bob' Martin definitely raises the bar with his latest book. He explains his expectation for a professional programmer on management interactions, time management, pressure, on collaboration, and on the choice of tools to use. Beyond TDD and ATDD, Martin explains what every programmer who considers him- or herself a professional not only needs to know, but also needs to follow in order to make the young profession of software development grow."

—Markus Gärtner
Senior Software Developer
it-agile GmbH
www.it-agile.de
www.shino.de

"Some technical books inspire and teach; some delight and amuse. Rarely does a technical book do all four of these things. Robert Martin's always have for me and *The Clean Coder* is no exception. Read, learn, and live the lessons in this book and you can accurately call yourself a software professional."

—George Bullock
Senior Program Manager
Microsoft Corp.

"If a computer science degree had 'required reading for after you graduate,' this would be it. In the real world, your bad code doesn't vanish when the semester's over, you don't get an A for marathon coding the night before an assignment's due, and, worst of all, you have to deal with people. So, coding gurus are not necessarily professionals. *The Clean Coder* describes the journey to professionalism . . . and it does a remarkably entertaining job of it."

—Jeff Overbey
University of Illinois at Urbana-Champaign

"*The Clean Coder* is much more than a set of rules or guidelines. It contains hard-earned wisdom and knowledge that is normally obtained through many years of trial and error or by working as an apprentice to a master craftsman. If you call yourself a software professional, you need this book."

—R. L. Bogetti
Lead System Designer
Baxter Healthcare
www.RLBogetti.com

This page intentionally left blank

The Clean Coder

The Clean Coder

A Code of Conduct for Professional Programmers

Robert C. Martin

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: www.informit.com/ph

Library of Congress Cataloging-in-Publication Data
Martin, Robert C.
The clean coder : a code of conduct for professional programmers / Robert Martin.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-708107-3 (pbk. : alk. paper)
1. Computer programming—Moral and ethical aspects. 2. Computer programmers—Professional ethics. I. Title.
QA76.9.M65M367 2011
005.1092—dc22 2011005962

ISBN-13: 978-0-13-708107-3
ISBN-10: 0-13-708107-3

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2011

Between 1986 and 2000 I worked closely with Jim Newkirk, a colleague from Teradyne. He and I shared a passion for programming and for clean code. We would spend nights, evenings, and weekends together playing with different programming styles and design techniques. We were continually scheming about business ideas. Eventually we formed Object Mentor, Inc., together. I learned many things from Jim as we plied our schemes together. But one of the most important was his attitude of *work ethic;* it was something I strove to emulate. Jim is a professional. I am proud to have worked with him, and to call him my friend.

This page intentionally left blank

Contents

Foreword

You've picked up this book, so I assume you are a software professional. That's good; so am I. And since I have your attention, let me tell you why I picked up this book.

It all starts a short time ago in a place not too far away. Cue the curtain, lights and camera, Charley ....

Several years ago I was working at a medium-sized corporation selling highly regulated products. You know the type; we sat in a cubicle farm in a three-story building, directors and up had private offices, and getting everyone you needed into the same room for a meeting took a week or so.

We were operating in a very competitive market when the government opened up a new product.

Suddenly we had an entirely new set of potential customers; all we had to do was to get them to buy our product. That meant we had to file by a certain deadline with the federal government, pass an assessment audit by another date, and go to market on a third date.

Over and over again our management stressed to us the importance of those dates. A single slip and the government would keep us out of the market for a year, and if customers couldn't sign up on day one, then they would all sign up with someone else and we'd be out of business.

It was the sort of environment in which some people complain, and others point out that "pressure makes diamonds."

I was a technical project manager, promoted from development. My responsibility was to get the web site up on go-live day, so potential customers could download information and, most importantly, enrollment forms. My partner in the endeavor was the business-facing project manager, whom I'll call Joe. Joe's role was to work the other side, dealing with sales, marketing, and the non-technical requirements. He was also the guy fond of the "pressure makes diamonds" comment.

If you've done much work in corporate America, you've probably seen the finger-pointing, blamestorming, and work aversion that is completely natural. Our company had an interesting solution to that problem with Joe and me.

A little bit like Batman and Robin, it was our job to get things done. I met with the technical team every day in a corner; we'd rebuild the schedule every single day, figure out the critical path, then remove every possible obstacle from that critical path. If someone needed software; we'd go get it. If they would "love to" configure the firewall but "gosh, it's time for my lunch break," we would buy them lunch. If someone wanted to work on our configuration ticket but had other priorities, Joe and I would go talk to the supervisor.

Then the manager.

Then the director.

We got things done.

It's a bit of an exaggeration to say that we kicked over chairs, yelled, and screamed, but we did use every single technique in our bag to get things done, invented a few new ones along the way, and we did it in an ethical way that I am proud of to this day.

I thought of myself as a member of the team, not above jumping in to write a SQL statement or doing a little pairing to get the code out the door. At the time, I thought of Joe the same way, as a member of the team, not above it.

Eventually I came to realize that Joe did not share that opinion. That was a very sad day for me.

It was Friday at 1:00 PM; the web site was set to go live very early the following Monday.

We were done. *DONE*. Every system was go; we were ready. I had the entire tech team assembled for the final scrum meeting and we were ready to flip the switch. More than "just" the technical team, we had the business folks from marketing, the product owners, with us.

We were proud. It was a good moment.

Then Joe dropped by.

He said something like, "Bad news. Legal doesn't have the enrollment forms ready, so we can't go live yet."

This was no big deal; we'd been held up by one thing or another for the length of the entire project and had the Batman/Robin routine down pat. I was ready, and my reply was essentially, "All right partner, let's do this one more time. Legal is on the third floor, right?"

Then things got weird.

Instead of agreeing with me, Joe asked, "What are you talking about Matt?"

I said, "You know. Our usual song and dance. We're talking about four PDF files, right? That are done; legal just has to approve them? Let's go hang out in their cubicles, give them the evil eye, and get this thing *done*!"

Joe did not agree with my assessment, and answered, "We'll just go live late next week. No big deal."

You can probably guess the rest of the exchange; it sounded something like this:

Matt:	“But why? They could do this in a couple hours.”
Joe:	“It might take more than that.”
Matt:	“But they’ve got *all weekend*. Plenty of time. Let’s do this!”
Joe:	“Matt, these are professionals. We can’t just stare them down and insist they sacrifice their personal lives for our little project.”
Matt:	(pause) “. . . Joe . . . what do you think we’ve been doing to the engineering team for the past four months?”
Joe:	“Yes, but these are professionals.”

Pause.

Breathe.

What. Did. Joe. Just. Say?

At the time, I thought the technical staff were professionals, in the best sense of the word.

Thinking back over it again, though, I’m not so sure.

Let’s look at that Batman and Robin technique a second time, from a different perspective. I thought I was exhorting the team to its best performance, but I suspect Joe was playing a game, with the implicit assumption that the technical staff was his opponent. Think about it: Why was it necessary to run around, kicking over chairs and leaning on people?

Shouldn’t we have been able to ask the staff when they would be done, get a firm answer, believe the answer we were given, and not be burned by that belief?

Certainly, for professionals, we should . . . and, at the same time, we could not. Joe didn’t trust our answers, and felt comfortable micromanaging the tech team—and at the same time, for some reason, he did trust the legal team and was not willing to micromanage them.

What's that all about?

Somehow, the legal team had demonstrated professionalism in a way the technical team had not.

Somehow, another group had convinced Joe that they did not need a babysitter, that they were not playing games, and that they needed to be treated as peers who were respected.

No, I don't think it had anything to do with fancy certificates hanging on walls or a few extra years of college, although those years of college might have included a fair bit of implicit social training on how to behave.

Ever since that day, those long years ago, I've wondered how the technical profession would have to change in order to be regarded as professionals.

Oh, I have a few ideas. I've blogged a bit, read a lot, managed to improve my own work life situation and help a few others. Yet I knew of no book that laid out a plan, that made the whole thing explicit.

Then one day, out of the blue, I got an offer to review an early draft of a book; the book that you are holding in your hands right now.

This book will tell step by step exactly how to present yourself and interact as a professional. Not with trite cliché, not with appeals to pieces of paper, but what you can do and how to do it.

In some cases, the examples are word for word.

Some of those examples have replies, counter-replies, clarifications, even advice for what to do if the other person tries to "just ignore you."

Hey, look at that, here comes Joe again, stage left this time:

Oh, here we are, back at BigCo, with Joe and me, once more on the big web site conversion project.

Only this time, imagine it just a little bit differently.

Instead of shirking from commitments, the technical staff actually makes them. Instead of shirking from estimates or letting someone else do the planning (then complaining about it), the technical team actually self-organizes and makes real commitments.

Now imagine that the staff is actually working together. When the programmers are blocked by operations, they pick up the phone and the sysadmin actually gets started on the work.

When Joe comes by to light a fire to get ticket 14321 worked on, he doesn't need to; he can see that the DBA is working diligently, not surfing the web. Likewise, the estimates he gets from staff seem downright consistent, and he doesn't get the feeling that the project is in priority somewhere between lunch and checking email. All the tricks and attempts to manipulate the schedule are not met with, "We'll try," but instead, "That's our commitment; if you want to make up your own goals, feel free."

After a while, I suspect Joe would start to think of the technical team as, well, professionals. And he'd be right.

Those steps to transform your behavior from technician to professional? You'll find them in the rest of the book.

Welcome to the next step in your career; I suspect you are going to like it.

—Matthew Heusser
Software Process Naturalist

Preface

At 11:39 AM EST on January 28, 1986, just 73.124 seconds after launch and at an altitude of 48,000 feet, the Space Shuttle Challenger was torn to smithereens by the failure of the right-hand solid rocket booster (SRB). Seven brave astronauts, including high school teacher Christa McAuliffe, were lost. The expression on the face of McAuliffe's mother as she watched the demise of her daughter nine miles overhead haunts me to this day.

The Challenger broke up because hot exhaust gasses in the failing SRB leaked out from between the segments of its hull, splashing across the body of the external fuel tank. The bottom of the main liquid hydrogen tank burst, igniting the fuel and driving the tank forward to smash into the liquid oxygen tank above it. At the same time the SRB detached from its aft strut and rotated around its forward strut. Its nose punctured the liquid oxygen tank. These aberrant force vectors caused the entire craft, moving well above mach 1.5, to rotate against the airstream. Aerodynamic forces quickly tore everything to shreds.

Between the circular segments of the SRB there were two concentric synthetic rubber O-rings. When the segments were bolted together the O-rings were compressed, forming a tight seal that the exhaust gasses should not have been able to penetrate.

But on the evening before the launch, the temperature on the launch pad got down to 17°F, 23 degrees below the O-rings' minimum specified temperature and 33 degrees lower than any previous launch. As a result, the O-rings grew too stiff to properly block the hot gasses. Upon ignition of the SRB there was a pressure pulse as the hot gasses rapidly accumulated. The segments of the booster ballooned outward and relaxed the compression on the O-rings. The stiffness of the O-rings prevented them from keeping the seal tight, so some of the hot gasses leaked through and vaporized the O-rings across 70 degrees of arc.

The engineers at Morton Thiokol who designed the SRB had known that there were problems with the O-rings, and they had reported those problems to managers at Morton Thiokol and NASA seven years earlier. Indeed, the O-rings from previous launches had been damaged in similar ways, though not enough to be catastrophic. The coldest launch had experienced the most damage. The engineers had designed a repair for the problem, but implementation of that repair had been long delayed.

The engineers suspected that the O-rings stiffened when cold. They also knew that temperatures for the Challenger launch were colder than any previous launch and well below the red-line. In short, the engineers *knew* that the risk was too high. The engineers acted on that knowledge. They wrote memos raising giant red flags. They strongly urged Thiokol and NASA managers not to launch. In an eleventh-hour meeting held just hours before the launch, those engineers presented their best data. They raged, and cajoled, and protested. But in the end, the managers ignored them.

When the time for launch came, some of the engineers refused to watch the broadcast because they feared an explosion on the pad. But as the Challenger climbed gracefully into the sky they began to relax. Moments before the destruction, as they watched the vehicle pass through Mach 1, one of them said that they'd "dodged a bullet."

Despite all the protest and memos, and urgings of the engineers, the managers believed they knew better. They thought the engineers were overreacting. They didn't trust the engineers' data or their conclusions. They launched because they were under immense financial and political pressure. They *hoped* everything would be just fine.

These managers were not merely foolish, they were criminal. The lives of seven good men and women, and the hopes of a generation looking toward space travel, were dashed on that cold morning because those managers set their own fears, hopes, and intuitions above the words of their own experts. They made a decision they had no right to make. They usurped the authority of the people who actually *knew*: the engineers.

But what about the engineers? Certainly the engineers did what they were supposed to do. They informed their managers and fought hard for their position. They went through the appropriate channels and invoked all the right protocols. They did what they could, *within* the system—and still the managers overrode them. So it would seem that the engineers can walk away without blame.

But sometimes I wonder whether any of those engineers lay awake at night, haunted by that image of Christa McAuliffe's mother, and wishing they'd called Dan Rather.

ABOUT THIS BOOK

This book is about software professionalism. It contains a lot of pragmatic advice in an attempt to answer questions, such as

- What is a software professional?
- How does a professional behave?
- How does a professional deal with conflict, tight schedules, and unreasonable managers?
- When, and how, should a professional say "no"?
- How does a professional deal with pressure?

But hiding within the pragmatic advice in this book you will find an attitude struggling to break through. It is an attitude of honesty, of honor, of self-respect, and of pride. It is a willingness to accept the dire responsibility of being a craftsman and an engineer. That responsibility includes working well and working clean. It includes communicating well and estimating faithfully. It includes managing your time and facing difficult risk-reward decisions.

But that responsibility includes one other thing—one frightening thing. As an engineer, you have a depth of knowledge about your systems and projects that no managers can possibly have. With that knowledge comes the responsibility to *act*.

BIBLIOGRAPHY

[**McConnell87**]**:** Malcolm McConnell, *Challenger 'A Major Malfunction'*, New York, NY: Simon & Schuster, 1987

[**Wiki-Challenger**]**: "**Space Shuttle Challenger disaster,"
http://en.wikipedia.org/wiki/Space_Shuttle_Challenger_disaster

Acknowledgments

My career has been a series of collaborations and schemes. Though I've had many private dreams and aspirations, I always seemed to find someone to share them with. In that sense I feel a bit like the Sith, "Always two there are."

The first collaboration that I could consider professional was with John Marchese at the age of 13. He and I schemed about building computers together. I was the brains and he was the brawn. I showed him where to solder a wire and he soldered it. I showed him where to mount a relay and he mounted it. It was a load of fun, and we spent hundreds of hours at it. In fact, we built quite a few very impressive-looking objects with relays, buttons, lights, even Teletypes! Of course, none of them actually did anything, but they were very impressive and we worked very hard on them. To John: Thank you!

In my freshman year of high school I met Tim Conrad in my German class. Tim was *smart*. When we teamed up to build a computer, he was the brains and I was the brawn. He taught me electronics and gave me my first introduction to a PDP-8. He and I actually built a working electronic 18-bit binary calculator out of basic components. It could add, subtract, multiply, and divide. It took us a year of weekends and all of spring, summer, and Christmas breaks. We worked furiously on it. In the end, it worked very nicely. To Tim: Thank you!

Tim and I learned how to program computers. This wasn't easy to do in 1968, but we managed. We got books on PDP-8 assembler, Fortran, Cobol, PL/1, among others. We devoured them. We wrote programs that we had no hope of executing because we did not have access to a computer. But we wrote them anyway for the sheer love of it.

Our high school started a computer science curriculum in our sophomore year. They hooked up an ASR-33 Teletype to a 110-baud, dial-up modem. They had an account on the Univac 1108 time-sharing system at the Illinois Institute of Technology. Tim and I immediately became the de facto operators of that machine. Nobody else could get near it.

The modem was connected by picking up the telephone and dialing the number. When you heard the answering modem squeal, you pushed the "orig" button on the Teletype causing the originating modem to emit its own squeal. Then you hung up the phone and the data connection was established.

The phone had a lock on the dial. Only the teachers had the key. But that didn't matter, because we learned that you could dial a phone (any phone) by tapping out the phone number on the switch hook. I was a drummer, so I had pretty good timing and reflexes. I could dial that modem, with the lock in place, in less than 10 seconds.

We had two Teletypes in the computer lab. One was the online machine and the other was an offline machine. Both were used by students to write their programs. The students would type their programs on the Teletypes with the paper tape punch engaged. Every keystroke was punched on tape. The students wrote their programs in IITran, a remarkably powerful interpreted language. Students would leave their paper tapes in a basket near the Teletypes.

After school, Tim and I would dial up the computer (by tapping of course), load the tapes into the IITran batch system, and then hang up. At 10 characters per second, this was not a quick procedure. An hour or so later, we'd call back and get the printouts, again at 10 characters per second. The Teletype did not separate the students' listings by ejecting pages. It just printed one after the next after the next, so we cut them apart using scissors, paper-clipped their input paper tape to their listing, and put them in the output basket.

Tim and I were the masters and gods of that process. Even the teachers left us alone when we were in that room. We were doing their job, and they knew it. They never asked us to do it. They never told us we could. They never gave us the key to the phone. We just moved in, and they moved out—and they gave us a very long leash. To my Math teachers, Mr. McDermit, Mr. Fogel, and Mr. Robien: Thank you!

Then, after all the student homework was done, we would play. We wrote program after program to do any number of mad and weird things. We wrote programs that graphed circles and parabolas in ASCII on a Teletype. We wrote random walk programs and random word generators. We calculated 50 factorial to the last digit. We spent hours and hours inventing programs to write and then getting them to work.

Two years later, Tim, our compadre Richard Lloyd, and I were hired as programmers at ASC Tabulating in Lake Bluff, Illinois. Tim and I were 18 at the time. We had decided that college was a waste of time and that we should begin our careers immediately. It was here that we met Bill Hohri, Frank Ryder, Big Jim Carlin, and John Miller. They gave some youngsters the opportunity to learn what professional programming was all about. The experience was not all positive and not all negative. It was certainly educational. To all of them, and to Richard who catalyzed and drove much of that process: Thank you.

After quitting and melting down at the age of 20, I did a stint as a lawn mower repairman working for my brother-in-law. I was so bad at it that he had to fire me. Thanks, Wes!

A year or so later I wound up working at Outboard Marine Corporation. By this time I was married and had a baby on the way. They fired me too. Thanks, John, Ralph, and Tom!

Then I went to work at Teradyne where I met Russ Ashdown, Ken Finder, Bob Copithorne, Chuck Studee, and CK Srithran (now Kris Iyer). Ken was my boss. Chuck and CK were my buds. I learned so much from all of them. Thanks, guys!

Then there was Mike Carew. At Teradyne, he and I became the dynamic duo. We wrote several systems together. If you wanted to get something done, and done fast, you got Bob and Mike to do it. We had a load of fun together. Thanks, Mike!

Jerry Fitzpatrick also worked at Teradyne. We met while playing Dungeons & Dragons together, but quickly formed a collaboration. We wrote software on a Commodore 64 to support D&D users. We also started a new project at Teradyne called "The Electronic Receptionist." We worked together for several years, and he became, and remains, a great friend. Thanks, Jerry!

I spent a year in England while working for Teradyne. There I teamed up with Mike Kergozou. He and I schemed together about all manner of things, though most of those schemes had to do with bicycles and pubs. But he was a dedicated programmer who was very focused on quality and discipline (though, perhaps he would disagree). Thanks, Mike!

Returning from England in 1987, I started scheming with Jim Newkirk. We both left Teradyne (months apart) and joined a start-up named Clear Communications. We spent several years together there toiling to make the millions that never came. But we continued our scheming. Thanks, Jim!

In the end we founded Object Mentor together. Jim is the most direct, disciplined, and focused person with whom I've ever had the privilege to work. He taught me so many things, I can't enumerate them here. Instead, I have dedicated this book to him.

There are so many others I've schemed with, so many others I've collaborated with, so many others who have had an impact on my professional life: Lowell Lindstrom, Dave Thomas, Michael Feathers, Bob Koss, Brett Schuchert, Dean Wampler, Pascal Roy, Jeff Langr, James Grenning, Brian Button, Alan Francis,

Mike Hill, Eric Meade, Ron Jeffries, Kent Beck, Martin Fowler, Grady Booch, and an endless list of others. Thank you, one and all.

Of course, the greatest collaborator of my life has been my lovely wife, Ann Marie. I married her when I was 20, three days after she turned 18. For 38 years she has been my steady companion, my rudder and sail, my love and my life. I look forward to another four decades with her.

And now, my collaborators and scheming partners are my children. I work closely with my eldest daughter Angela, my lovely mother hen and intrepid assistant. She keeps me on the straight and narrow and never lets me forget a date or commitment. I scheme business plans with my son Micah, the founder of 8thlight.com. His head for business is far better than mine ever was. Our latest venture, cleancoders.com, is very exciting!

My younger son Justin has just started working with Micah at 8th Light. My younger daughter Gina is a chemical engineer working for Honeywell. With those two, the serious scheming has just begun!

No one in your life will teach you more than your children will. Thanks, kids!

This page intentionally left blank

About the Author

Robert C. Martin (**"Uncle Bob"**) has been a programmer since 1970. He is founder and president of Object Mentor, Inc., an international firm of highly experienced software developers and managers who specialize in helping companies get their projects done. Object Mentor offers process improvement consulting, object-oriented software design consulting, training, and skill development services to major corporations worldwide.

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:

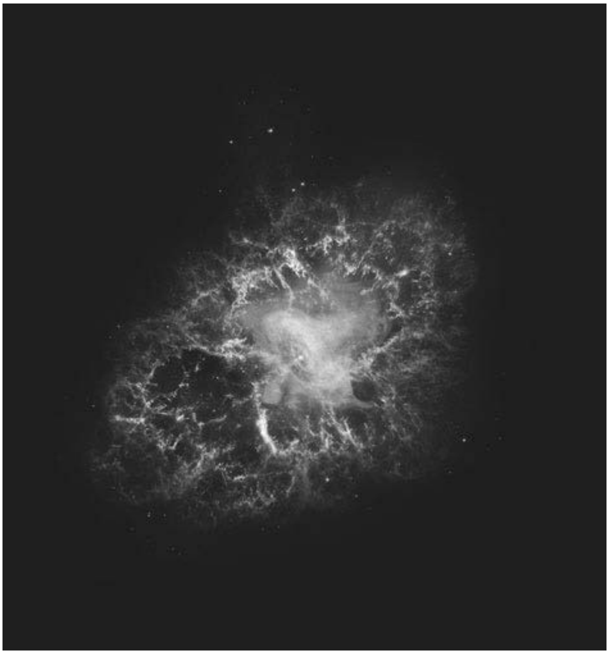
- *Designing Object Oriented C++ Applications Using the Booch Method*
- *Patterns Languages of Program Design 3*

- *More C++ Gems*
- *Extreme Programming in Practice*
- *Agile Software Development: Principles, Patterns, and Practices*
- *UML for Java Programmers*
- *Clean Code*

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.

Robert is also the founder of Uncle Bob Consulting, LLC, and cofounder with his son Micah Martin of The Clean Coders LLC.

On the Cover



The stunning image on the cover, reminiscent of Sauron's eye, is M1, the Crab Nebula. M1 is located in Taurus, about one degree to the right of Zeta Tauri, the star at the tip of the bull's left horn. The crab nebula is the remnant of a supernova that blew its guts all over the sky on the rather auspicious date of July 4th, 1054 AD. At a distance of 6500 light years, that explosion appeared to Chinese observers as a new star, roughly as bright as Jupiter. Indeed, it was visible *during the day*! Over the next six months it slowly faded from naked-eye view.

The cover image is a composite of visible and x-ray light. The visible image was taken by the Hubble telescope and forms the outer envelope. The inner object that looks like a blue archery target was taken by the Chandra x-ray telescope.

The visible image depicts a rapidly expanding cloud of dust and gas laced with heavy elements left over from the supernova explosion. That cloud is now 11 light-years in diameter, weighs in at 4.5 solar masses, and is expanding at the furious rate of 1500 kilometers per second. The kinetic energy of that old explosion is impressive to say the least.

At the very center of the target is a bright blue dot. That's where the *pulsar* is. It was the formation of the pulsar that caused the star to blow up in the first place. Nearly a solar mass of material in the core of the doomed star imploded into a sphere of neutrons about 30 kilometers in diameter. The kinetic energy of that implosion, coupled with the incredible barrage of neutrinos created when all those neutrons formed, ripped the star open, and blew it to kingdom come.

The pulsar is spinning about 30 times per second; and it flashes as it spins. We can see it blinking in our telescopes. Those pulses of light are the reason we call it a pulsar, which is short for Pulsating Star.

Pre-Requisite Introduction

(Don't skip this, you're going to need it.)

I presume you just picked up this book because you are a computer programmer and are intrigued by the notion of professionalism. You should be. Professionalism is something that our profession is in dire need of.

I'm a programmer too. I've been a programmer for 42[1] years; and in that time—*let me tell you*—I've seen it all. I've been fired. I've been lauded. I've been a team leader, a manager, a grunt, and even a CEO. I've worked with brilliant

1. Don't Panic.

programmers and I've worked with slugs.[2] I've worked on high-tech cutting-edge embedded software/hardware systems, and I've worked on corporate payroll systems. I've programmed in COBOL, FORTRAN, BAL, PDP-8, PDP-11, C, C++, Java, Ruby, Smalltalk, and a plethora of other languages and systems. I've worked with untrustworthy paycheck thieves, and I've worked with consummate professionals. It is that last classification that is the topic of this book.

In the pages of this book I will try to define what it means to be a professional programmer. I will describe the attitudes, disciplines, and actions that I consider to be essentially professional.

How do I know what these attitudes, disciplines, and actions are? Because I had to learn them the hard way. You see, when I got my first job as a programmer, professional was the last word you'd have used to describe me.

The year was 1969. I was 17. My father had badgered a local business named ASC into hiring me as a temporary part-time programmer. (Yes, my father could do things like that. I once watched him walk out in front of a speeding car with his hand out commanding it to "Stop!" The car stopped. Nobody said "no" to my Dad.) The company put me to work in the room where all the IBM computer manuals were kept. They had me put years and years of updates into the manuals. It was here that I first saw the phrase: "This page intentionally left blank."

After a couple of days of updating manuals, my supervisor asked me to write a simple Easycoder[3] program. I was thrilled to be asked. I'd never written a program for a real computer before. I had, however, inhaled the Autocoder books, and had a vague notion of how to begin.

The program was simply to read records from a tape, and replace the IDs of those records with new IDs. The new IDs started at 1 and were incremented by

2. A technical term of unknown origins.
3. Easycoder was the assembler for the Honeywell H200 computer, which was similar to Autocoder for the IBM 1401 computer.

1 for each new record. The records with the new IDs were to be written to a new tape.

My supervisor showed me a shelf that held many stacks of red and blue punched cards. Imagine that you bought 50 decks of playing cards, 25 red decks, and 25 blue decks. Then you stacked those decks one on top of the other. That's what these stacks of cards looked like. They were striped red and blue, and the stripes were about 200 cards each. Each one of those stripes contained the source code for the subroutine library that the programmers typically used. Programmers would simply take the top deck off the stack, making sure that they took nothing but red or blue cards, and then put that at the end of their program deck.

I wrote my program on some coding forms. Coding forms were large rectangular sheets of paper divided into 25 lines and 80 columns. Each line represented one card. You wrote your program on the coding form using block capital letters and a #2 pencil. In the last 6 columns of each line you wrote a sequence number with that #2 pencil. Typically you incremented the sequence number by 10 so that you could insert cards later.

The coding form went to the key punchers. This company had several dozen women who took coding forms from a big in-basket, and then "typed" them into key-punch machines. These machines were a lot like typewriters, except that the characters were punched into cards instead of printed on paper.

The next day the keypunchers returned my program to me by inter-office mail. My small deck of punched cards was wrapped up by my coding forms and a rubber band. I looked over the cards for keypunch errors. There weren't any. So then I put the subroutine library deck on the end of my program deck, and then took the deck upstairs to the computer operators.

The computers were behind locked doors in an environmentally controlled room with a raised floor (for all the cables). I knocked on the door and an operator austerely took my deck from me and put it into another in-basket inside the computer room. When they got around to it, they would run my deck.

The next day I got my deck back. It was wrapped in a listing of the results of the run and kept together with a rubber band. (We used *lots* of rubber bands in those days!)

I opened the listing and saw that my compile had failed. The error messages in the listing were very difficult for me to understand, so I took it to my supervisor. He looked it over, mumbled under his breath, made some quick notes on the listing, grabbed my deck and then told me to follow him.

He took me up to the keypunch room and sat at a vacant keypunch machine. One by one he corrected the cards that were in error, and added one or two other cards. He quickly explained what he was doing, but it all went by like a flash.

He took the new deck up to the computer room and knocked at the door. He said some magic words to one of the operators, and then walked into the computer room behind him. He beckoned for me to follow. The operator set up the tape drives and loaded the deck while we watched. The tapes spun, the printer chattered, and then it was over. The program had worked.

The next day my supervisor thanked me for my help, and terminated my employment. Apparently ASC didn't feel they had the time to nurture a 17-year-old.

But my connection with ASC was hardly over. A few months later I got a full-time second-shift job at ASC operating off-line printers. These printers printed junk mail from print images that were stored on tape. My job was to load the printers with paper, load the tapes into the tape drives, fix paper jams, and otherwise just watch the machines work.

The year was 1970. College was not an option for me, nor did it hold any particular enticements. The Viet Nam war was still raging, and the campuses were chaotic. I had continued to inhale books on COBOL, Fortran, PL/1, PDP-8, and IBM 360 Assembler. My intent was to bypass school and drive as hard as I could to get a job programming.

Twelve months later I achieved that goal. I was promoted to a full-time programmer at ASC. I, and two of my good friends, Richard and Tim, also 19, worked with a team of three other programmers writing a real-time accounting system for a teamster's union. The machine was a Varian 620i. It was a simple mini-computer similar in architecture to a PDP-8 except that it had a 16-bit word and two registers. The language was assembler.

We wrote every line of code in that system. And I mean *every* line. We wrote the operating system, the interrupt heads, the IO drivers, the *file system* for the disks, the overlay swapper, and even the relocatable linker. Not to mention all the application code. We wrote all this in 8 months working 70 and 80 hours a week to meet a hellish deadline. My salary was $7,200 per year.

We delivered that system. And then we quit.

We quit suddenly, and with malice. You see, after all that work, and after having delivered a successful system, the company gave us a 2% raise. We felt cheated and abused. Several of us got jobs elsewhere and simply resigned.

I, however, took a different, and very unfortunate, approach. I and a buddy stormed into the boss' office and quit together rather loudly. This was emotionally very satisfying—for a day.

The next day it hit me that I did not have a job. I was 19, unemployed, with no degree. I interviewed for a few programming positions, but those interviews did not go well. So I worked in my brother-in-law's lawnmower repair shop for four months. Unfortunately I was a lousy lawnmower repairman. He eventually had to let me go. I fell into a nasty funk.

I stayed up till 3 AM every night eating pizza and watching old monster movies on my parents' old black-and-white, rabbit-ear TV. Only some of the ghosts where characters in the movies. I stayed in bed till 1 PM because I didn't want to face my dreary days. I took a calculus course at a local community college and failed it. I was a wreck.

My mother took me aside and told me that my life was a mess, and that I had been an idiot for quitting without having a new job, and for quitting so emotionally, and for quitting together with my buddy. She told me that you never quit without having a new job, and you always quit calmly, coolly, and alone. She told me that I should call my old boss and beg for my old job back. She said, “You need to eat some humble pie.”

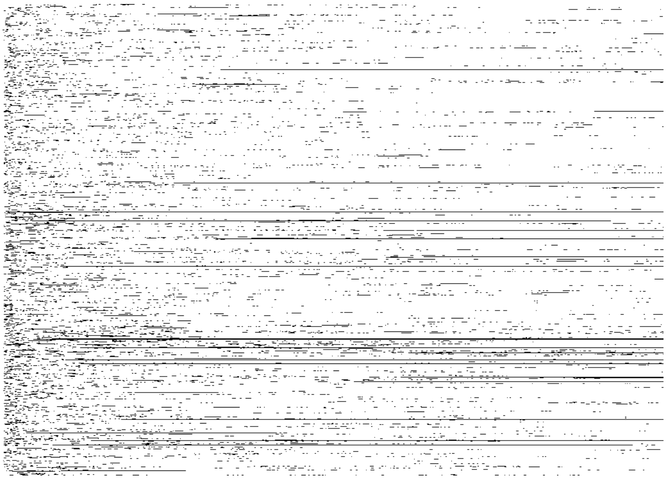
Nineteen-year-old boys are not known for their appetite for humble pie, and I was no exception. But the circumstances had taken their toll on my pride. In the end I called my boss and took a big bite of that humble pie. And it worked. He was happy to re-hire me for $6,800 per year, and I was happy to take it.

I spent another eighteen months working there, watching my Ps and Qs and trying to be as valuable an employee as I could. I was rewarded with promotions and raises, and a regular paycheck. Life was good. When I left that company, it was on good terms, and with an offer for a better job in my pocket.

You might think that I had learned my lesson; that I was now a professional. Far from it. That was just the first of many lessons I needed to learn. In the coming years I would be fired from one job for carelessly missing critical dates, and nearly fired from still another for inadvertently leaking confidential information to a customer. I would take the lead on a doomed project and ride it into the ground without calling for the help I knew I needed. I would aggressively defend my technical decisions even though they flew in the face of the customers’ needs. I would hire one wholly unqualified person, saddling my employer with a huge liability to deal with. And worst of all, I would get two other people fired because of my inability to lead.

So think of this book as a catalog of my own errors, a blotter of my own crimes, and a set of guidelines for you to avoid walking in my early shoes.

4 CODING



In a previous book[1] I wrote a great deal about the structure and nature of *Clean Code*. This chapter discusses the *act* of coding, and the context that surrounds that act.

When I was 18 I could type reasonably well, but I had to look at the keys. I could not type blind. So one evening I spent a few long hours at an IBM 029 keypunch refusing to look at my fingers as I typed a program that I had written on several coding forms. I examined each card after I typed it and discarded those that were typed wrong.

1. [Martin09]

At first I typed quite a few in error. By the end of the evening I was typing them all with near perfection. I realized, during that long night, that typing blind is all about *confidence*. My fingers knew where the keys were, I just had to gain the confidence that I wasn't making a mistake. One of the things that helped with that confidence is that I could *feel* when I was making an error. By the end of the evening, if I made a mistake, I knew it almost instantly and simply ejected the card without looking at it.

Being able to sense your errors is really important. Not just in typing, but in everything. Having error-sense means that you very rapidly close the feedback loop and learn from your errors all the more quickly. I've studied, and mastered, several disciplines since that day on the 029. I've found that in each case that the key to mastery is confidence and error-sense.

This chapter describes my personal set of rules and principles for coding. These rules and principles are not about my code itself; they are about my behavior, mood, and attitude while writing code. They describe my own mental, moral, and emotional context for writing code. These are the roots of my confidence and error-sense.

You will likely not agree with everything I say here. After all, this is deeply personal stuff. In fact, you may violently disagree with some of my attitudes and principles. That's OK—they are not intended to be absolute truths for anyone other than me. What they are is one man's approach to being a professional coder.

Perhaps, by studying and contemplating my own personal coding milieu you can learn to snatch the pebble from my hand.

Preparedness

Coding is an intellectually challenging and exhausting activity. It requires a level of concentration and focus that few other disciplines require. The reason for this is that coding requires you to juggle many competing factors at once.

1. First, your code must work. You must understand what problem you are solving and understand how to solve that problem. You must ensure that the code you write is a faithful representation of that solution. You must manage

every detail of that solution while remaining consistent within the language, platform, current architecture, and all the warts of the current system.

2. Your code must solve the problem set for you by the customer. Often the customer's requirements do not actually solve the customer's problems. It is up to you to see this and negotiate with the customer to ensure that the customer's true needs are met.
3. Your code must fit well into the existing system. It should not increase the rigidity, fragility, or opacity of that system. The dependencies must be well-managed. In short, your code needs to follow solid engineering principles.[2]
4. Your code must be readable by other programmers. This is not simply a matter of writing nice comments. Rather, it requires that you craft the code in such a way that it reveals your intent. This is hard to do. Indeed, this may be the most difficult thing a programmer can master.

Juggling all these concerns is hard. It is physiologically difficult to maintain the necessary concentration and focus for long periods of time. Add to this the problems and distractions of working in a team, in an organization, and the cares and concerns of everyday life. The bottom line is that the opportunity for distraction is high.

When you cannot concentrate and focus sufficiently, the code you write will be wrong. It will have bugs. It will have the wrong structure. It will be opaque and convoluted. It will not solve the customers' real problems. In short, it will have to be reworked or redone. Working while distracted creates waste.

If you are tired or distracted, *do not code*. You'll only wind up redoing what you did. Instead, find a way to eliminate the distractions and settle your mind.

3 AM CODE

The worst code I ever wrote was at 3 AM. The year was 1988, and I was working at a telecommunications start-up named Clear Communications. We were all putting in long hours in order to build "sweat equity." We were, of course, all dreaming of being rich.

2. [Martin03]

One very late evening—or rather, one very early morning, in order to solve a timing problem—I had my code send a message to itself through the event dispatch system (we called this “sending mail”). This was the *wrong* solution, but at 3 AM it looked pretty damned good. Indeed, after 18 hours of solid coding (not to mention the 60–70 hour weeks) it was *all* I could think of.

I remember feeling so good about myself for the long hours I was working. I remember feeling *dedicated*. I remember thinking that working at 3 AM is what serious professionals do. How wrong I was!

That code came back to bite us over and over again. It instituted a faulty design structure that everyone used but consistently had to work around. It caused all kinds of strange timing errors and odd feedback loops. We’d get into infinite mail loops as one message caused another to be sent, and then another, infinitely. We never had time to rewrite this wad (so we thought) but we always seemed to have time to add another wart or patch to work around it. The cruft grew and grew, surrounding that 3 AM code with ever more baggage and side effects. Years later it had become a team joke. Whenever I was tired or frustrated they’d say, “Look out! Bob’s about to send mail to himself!”

The moral of this story is: Don’t write code when you are tired. Dedication and professionalism are more about discipline than hours. Make sure that your sleep, health, and lifestyle are tuned so that you can put in eight *good* hours per day.

Worry Code

Have you ever gotten into a big fight with your spouse or friend, and then tried to code? Did you notice that there was a background process running in your mind trying to resolve, or at least review the fight? Sometimes you can feel the stress of that background process in your chest, or in the pit of your stomach. It can make you feel anxious, like when you’ve had too much coffee or diet coke. It’s distracting.

When I am worried about an argument with my wife, or a customer crisis, or a sick child, I can’t maintain focus. My concentration wavers. I find myself with my eyes on the screen and my fingers on the keyboard, doing nothing. Catatonic.

Paralyzed. A million miles away working through the problem in the background rather than actually solving the coding problem in front of me.

Sometimes I will force myself to *think* about the code. I might drive myself to write a line or two. I might push myself to get a test or two to pass. But I can't keep it up. Inevitably I find myself descending into a stupefied insensibility, seeing nothing through my open eyes, inwardly churning on the background worry.

I have learned that this is no time to code. Any code I produce will be trash. So instead of coding, I need to resolve the worry.

Of course, there are many worries that simply cannot be resolved in an hour or two. Moreover, our employers are not likely to long tolerate our inability to work as we resolve our personal issues. The trick is to learn how to shut down the background process, or at least reduce its priority so that it's not a continuous distraction.

I do this by partitioning my time. Rather than forcing myself to code while the background worry is nagging at me, I will spend a dedicated block of time, perhaps an hour, working on the issue that is creating the worry. If my child is sick, I will call home and check in. If I've had an argument with my wife, I'll call her and talk through the issues. If I have money problems, I'll spend time thinking about how I can deal with the financial issues. I know I'm not likely to solve the problems in this hour, but it is very likely that I can reduce the anxiety and quiet the background process.

Ideally the time spent wrestling with personal issues would be personal time. It would be a shame to spend an hour at the office this way. Professional developers allocate their personal time in order to ensure that the time spent at the office is as productive as possible. That means you should specifically set aside time at home to settle your anxieties so that you don't bring them to the office.

On the other hand, if you find yourself at the office and the background anxieties are sapping your productivity, then it is better to spend an hour quieting them than to use brute force to write code that you'll just have to throw away later (or worse, live with).

The Flow Zone

Much has been written about the hyper-productive state known as "flow." Some programmers call it "the Zone." Whatever it is called, you are probably familiar with it. It is the highly focused, tunnel-vision state of consciousness that programmers can get into while they write code. In this state they feel *productive*. In this state they feel *infallible*. And so they desire to attain that state, and often measure their self-worth by how much time they can spend there.

Here's a little hint from someone whose been there and back: *Avoid the Zone*. This state of consciousness is not really hyper-productive and is certainly not infallible. It's really just a mild meditative state in which certain rational faculties are diminished in favor of a sense of speed.

Let me be clear about this. You *will* write more code in the Zone. If you are practicing TDD, you will go around the red/green/refactor loop more quickly. And you will *feel* a mild euphoria or a sense of conquest. The problem is that you lose some of the big picture while you are in the Zone, so you will likely make decisions that you will later have to go back and reverse. Code written in the Zone may come out faster, but you'll be going back to visit it more.

Nowadays when I feel myself slipping into the Zone, I walk away for a few minutes. I clear my head by answering a few emails or looking at some tweets. If it's close enough to noon, I'll break for lunch. If I'm working on a team, I'll find a pair partner.

One of the big benefits of pair programming is that it is virtually impossible for a pair to enter the Zone. The Zone is an uncommunicative state, while pairing requires intense and constant communication. Indeed, one of the complaints I often hear about pairing is that it blocks entry into the Zone. Good! The Zone is *not* where you want to be.

Well, that's not *quite* true. There are times when the Zone is exactly where you want to be. When you are *practicing*. But we'll talk about that in another chapter.

MUSIC

At Teradyne, in the late '70s, I had a private office. I was the system administrator of our PDP 11/60, and so I was one of the few programmers allowed to have a private terminal. That terminal was a VT100 running at 9600 baud and connected to the PDP 11 with 80 feet of RS232 cable that I had strung over the ceiling tiles from my office to the computer room.

I had a stereo system in my office. It was an old turntable, amp, and floor speakers. I had a significant collection of vinyl, including Led Zeppelin, Pink Floyd, and …. Well, you get the picture.

I used to crank that stereo and then write code. I thought it helped my concentration. But I was wrong.

One day I went back into a module that I had been editing while listening to the opening sequence of *The Wall*. The comments in that code contained lyrics from the piece, and editorial notations about dive bombers and crying babies.

That's when it hit me. As a reader of the code, I was learning more about the music collection of the author (me) than I was learning about the problem that the code was trying to solve.

I realized that I simply don't code well while listening to music. The music does not help me focus. Indeed, the act of listening to music seems to consume some vital resource that my mind needs in order to write clean and well-designed code.

Maybe it doesn't work that way for you. Maybe music *helps* you write code. I know lots of people who code while wearing earphones. I accept that the music may help them, but I am also suspicious that what's really happening is that the music is helping them enter the Zone.

INTERRUPTIONS

Visualize yourself as you are coding at your workstation. How do you respond when someone asks you a question? Do you snap at them? Do you glare? Does your body-language tell them to go away because you are busy? In short, are you rude?

Or, do you stop what you are doing and politely help someone who is stuck? Do you treat them as you would have them treat you if you were stuck?

The rude response often comes from the Zone. You may resent being dragged out of the Zone, or you may resent someone interfering with your attempt to enter the Zone. Either way, the rudeness often comes from your relationship to the Zone.

Sometimes, however, it's not the Zone that's at fault, it's just that you are trying to understand something complicated that requires concentration. There are several solutions to this.

Pairing can be very helpful as a way to deal with interruptions. Your pair partner can hold the context of the problem at hand, while you deal with a phone call, or a question from a coworker. When you return to your pair partner, he quickly helps you reconstruct the mental context you had before the interruption.

TDD is another big help. If you have a failing test, that test holds the context of where you are. You can return to it after an interruption and continue to make that failing test pass.

In the end, of course, *there will be interruptions* that distract you and cause you to lose time. When they happen, remember that next time you may be the one who needs to interrupt someone else. So the professional attitude is a polite willingness to be helpful.

Writer's Block

Sometimes the code just doesn't come. I've had this happen to me and I've seen it happen to others. You sit at your workstation and nothing happens.

Often you will find other work to do. You'll read email. You'll read tweets. You'll look through books, or schedules, or documents. You'll call meetings. You'll start up conversations with others. You'll do *anything* so that you don't have to face that workstation and watch as the code refuses to appear.

What causes such blockages? We've spoken about many of the factors already. For me, another major factor is sleep. If I'm not getting enough sleep, I simply can't code. Others are worry, fear, and depression.

Oddly enough there is a very simple solution. It works almost every time. It's easy to do, and it can provide you with the momentum to get lots of code written.

The solution: Find a pair partner.

It's uncanny how well this works. As soon as you sit down next to someone else, the issues that were blocking you melt away. There is a *physiological* change that takes place when you work with someone. I don't know what it is, but I can definitely feel it. There's some kind of chemical change in my brain or body that breaks me through the blockage and gets me going again.

This is not a perfect solution. Sometimes the change lasts an hour or two, only to be followed by exhaustion so severe that I have to break away from my pair partner and find some hole to recover in. Sometimes, even when sitting with someone, I can't do more than just agree with what that person is doing. But for me the typical reaction to pairing is a recovery of my momentum.

Creative Input

There are other things I do to prevent blockage. I learned a long time ago that creative output depends on creative input.

I read a lot, and I read all kinds of material. I read material on software, politics, biology, astronomy, physics, chemistry, mathematics, and much more. However, I find that the thing that best primes the pump of creative output is science fiction.

For you, it might be something else. Perhaps a good mystery novel, or poetry, or even a romance novel. I think the real issue is that creativity breeds creativity. There's also an element of escapism. The hours I spend away from my usual problems, while being actively stimulated by challenging and creative ideas, results in an almost irresistible pressure to create something myself.

Not all forms of creative input work for me. Watching TV does not usually help me create. Going to the movies is better, but only a bit. Listening to music does not help me create code, but does help me create presentations, talks, and videos. Of all the forms of creative input, nothing works better for me than good old space opera.

Debugging

One of the worst debugging sessions in my career happened in 1972. The terminals connected to the Teamsters' accounting system used to freeze once or twice a day. There was no way to force this to happen. The error did not prefer any particular terminals or any particular applications. It didn't matter what the user had been doing before the freeze. One minute the terminal was working fine, and the next minute it was hopelessly frozen.

It took weeks to diagnose this problem. Meanwhile the Teamsters' were getting more and more upset. Every time there was a freeze-up the person at that terminal would have to stop working and wait until they could coordinate all the other users to finish their tasks. Then they'd call us and we'd reboot. It was a nightmare.

We spent the first couple of weeks just gathering data by interviewing the people who experienced the lockups. We'd ask them what they were doing at the time, and what they had done previously. We asked other users if they noticed anything on *their* terminals at the time of the freeze-up. These interviews were all done over the phone because the terminals were located in downtown Chicago, while we worked 30 miles north in the cornfields.

We had no logs, no counters, no debuggers. Our only access to the internals of the system were lights and toggle switches on the front panel. We could stop the computer, and then peek around in memory one word at a time. But we couldn't do this for more than five minutes because the Teamsters' needed their system back up.

We spent a few days writing a simple real-time inspector that could be operated from the ASR-33 teletype that served as our console. With this we could peek and poke around in memory while the system was running. We added log messages that printed on the teletype at critical moments. We created in-memory counters that counted events and remembered state history that we could inspect with the inspector. And, of course, all this had to be written from scratch in assembler and tested in the evenings when the system was not in use.

The terminals were interrupt driven. The characters being sent to the terminals were held in circular buffers. Every time a serial port finished sending a character, an interrupt would fire and the next character in the circular buffer would be readied for sending.

We eventually found that when a terminal froze it was because the three variables that managed the circular buffer were out of sync. We had no idea why this was happening, but at least it was a clue. Somewhere in the 5 KSLOC of supervisory code there was a bug that mishandled one of those pointers.

This new knowledge also allowed us to un-freeze terminals manually! We could poke default values into those three variables using the inspector, and the terminals would magically start running again. Eventually we wrote a little hack that would look through all the counters to see if they were misaligned and repair them. At first we invoked that hack by hitting a special user-interrupt switch on the front panel whenever the Teamsters called to report a freeze-up. Later we simply ran the repair utility once every second.

A month or so later the freeze-up issue was dead, as far as the Teamsters were concerned. Occasionally one of their terminals would pause for a half second or so, but at a base rate of 30 characters per second, nobody seemed to notice.

But why were the counters getting misaligned? I was nineteen and determined to find out.

The supervisory code had been written by Richard, who had since gone off to college. None of the rest of us were familiar with that code because Richard had been quite possessive of it. That code was *his*, and we weren't allowed to know it. But now Richard was gone, so I got out the inches-thick listing and started to go over it page by page.

The circular queues in that system were just FIFO data structures, that is, queues. Application programs pushed characters in one end of the queue until the queue was full. The interrupt heads popped the characters off the other end of the queue when the printer is ready for them. When the queue was empty, the printer would stop. Our bug caused the applications to think that the queue was full, but caused the interrupt heads to think that the queue was empty.

Interrupt heads run in a different "thread" than all other code. So counters and variables that are manipulated by both interrupt heads and other code must be protected from concurrent update. In our case that meant turning the interrupts off around any code that manipulated those three variables. By the time I sat down with that code I knew I was looking for someplace in the code that touched the variables but did not disable the interrupts first.

Nowadays, of course, we'd use the plethora of powerful tools at our disposal to find all the places where the code touched those variables. Within seconds we'd know every line of code that touched them. Within minutes we'd know which did not disable the interrupts. But this was 1972, and I didn't have any tools like that. What I had were my eyes.

I pored over every page of that code, looking for the variables. Unfortunately, the variables were used *everywhere*. Nearly every page touched them in one way or another. Many of those references did not disable the interrupts because they were read-only references and therefore harmless. The problem was, in that particular assembler there was no good way to know if a reference was read-only without following the logic of the code. Any time a variable was read, it might later be updated and stored. And if that happened while the interrupts were enabled, the variables could get corrupted.

It took me days of intense study, but in the end I found it. There, in the middle of the code, was one place where one of the three variables was being updated while the interrupts were enabled.

I did the math. The vulnerability was about two microseconds long. There were a dozen terminals all running at 30 cps, so an interrupt every 3 ms or so. Given the size of the supervisor, and the clock rate of the CPU, we'd expect a freeze-up from this vulnerability one or two times a day. Bingo!

I fixed the problem, of course, but never had the courage to turn off the automatic hack that inspected and fixed the counters. To this day I'm not convinced there wasn't another hole.

DEBUGGING TIME

For some reason software developers don't think of debugging time as coding time. They think of debugging time as a call of nature, something that just *has* to be done. But debugging time is just as expensive to the business as coding time is, and therefore anything we can do to avoid or diminish it is good.

Nowadays I spend much less time debugging than I did ten years ago. I haven't measured the difference, but I believe it's about a factor of ten. I achieved this truly radical reduction in debugging time by adopting the practice of Test Driven Development (TDD), which we'll be discussing in another chapter.

Whether you adopt TDD or some other discipline of equal efficacy,[3] it is incumbent upon you as a professional to reduce your debugging time as close to zero as you can get. Clearly zero is an asymptotic goal, but it is the goal nonetheless.

Doctors don't like to reopen patients to fix something they did wrong. Lawyers don't like to retry cases that they flubbed up. A doctor or lawyer who did that too often would not be considered professional. Likewise, a software developer who creates many bugs is acting unprofessionally.

PACING YOURSELF

Software development is a marathon, not a sprint. You can't win the race by trying to run as fast as you can from the outset. You win by conserving your resources and pacing yourself. A marathon runner takes care of her body both before and *during* the race. Professional programmers conserve their energy and creativity with the same care.

3. I don't know of any discipline that is as effective as TDD, but perhaps you do.

Know When to Walk Away

Can't go home till you solve this problem? Oh yes you can, and you probably should! Creativity and intelligence are fleeting states of mind. When you are tired, they go away. If you then pound your nonfunctioning brain for hour after late-night hour trying to solve a problem, you'll simply make yourself more tired and reduce the chance that the shower, or the car, will help you solve the problem.

When you are stuck, when you are tired, disengage for awhile. Give your creative subconscious a crack at the problem. You will get more done in less time and with less effort if you are careful to husband your resources. Pace yourself, and your team. Learn your patterns of creativity and brilliance, and take advantage of them rather than work against them.

Driving Home

One place that I have solved a number of problems is my car on the way home from work. Driving requires a lot of noncreative mental resources. You must dedicate your eyes, hands, and portions of your mind to the task; therefore, you must disengage from the problems at work. There is something about *disengagement* that allows your mind to hunt for solutions in a different and more creative way.

The Shower

I have solved an inordinate number of problems in the shower. Perhaps that spray of water early in the morning wakes me up and gets me to review all the solutions that my brain came up with while I was asleep.

When you are working on a problem, you sometimes get so close to it that you can't see all the options. You miss elegant solutions because the creative part of your mind is suppressed by the intensity of your focus. Sometimes the best way to solve a problem is to go home, eat dinner, watch TV, go to bed, and then wake up the next morning and take a shower.

Being Late

You *will* be late. It happens to the best of us. It happens to the most dedicated of us. Sometimes we just blow our estimates and wind up late.

The trick to managing lateness is early detection and transparency. The worst case scenario occurs when you continue to tell everyone, up to the very end, that you will be on time—and then let them all down. *Don't* do this. Instead, *regularly* measure your progress against your goal, and come up with three[4] fact-based end dates: best case, nominal case, and worst case. Be as honest as you can about all three dates. *Do not incorporate hope into your estimates!* Present all three numbers to your team and stakeholders. Update these numbers daily.

Hope

What if these numbers show that you *might* miss a deadline? For example, let's say that there's a trade show in ten days, and we need to have our product there. But let's also say that your three-number estimate for the feature you are working on is 8/12/20.

Do not hope that you can get it all done in ten days! Hope is the project killer. Hope destroys schedules and ruins reputations. Hope will get you into deep trouble. If the trade show is in ten days, and your nominal estimate is 12, you *are not* going to make it. Make sure that the team and the stakeholders understand the situation, and don't let up until there is a fall-back plan. Don't let anyone else have hope.

Rushing

What if your manager sits you down and asks you to try to make the deadline? What if your manager insists that you "do what it takes"? *Hold to your estimates!* Your original estimates are more accurate than any changes you make while

4. There's much more about this in the Estimation chapter.

your boss is confronting you. Tell your boss that you've already considered the options (because you have) and that the only way to improve the schedule is to reduce scope. *Do not be tempted to rush.*

Woe to the poor developer who buckles under pressure and agrees to *try* to make the deadline. That developer will start taking shortcuts and working extra hours in the vain hope of working a miracle. This is a recipe for disaster because it gives you, your team, and your stakeholders false hope. It allows everyone to avoid facing the issue and delays the necessary tough decisions.

There is no way to rush. You can't make yourself code faster. You can't make yourself solve problems faster. If you try, you'll just slow yourself down and make a mess that slows everyone else down, too.

So you must answer your boss, your team, and your stakeholders by depriving them of hope.

Overtime

So your boss says, "What if you work an extra two hours a day? What if you work on Saturday? Come on, there's just got to be a way to squeeze enough hours in to get the feature done on time."

Overtime can work, and sometimes it is necessary. Sometimes you can make an otherwise impossible date by putting in some ten-hour days, and a Saturday or two. But this is very risky. You are not likely to get 20% more work done by working 20% more hours. What's more, overtime will *certainly* fail if it goes on for more than two or three weeks.

Therefore you should *not* agree to work overtime unless (1) you can personally afford it, (2) it is short term, two weeks or less, and (3) *your boss has a fall-back plan* in case the overtime effort fails.

That last criterion is a deal breaker. If your boss cannot articulate to you what he's going to do if the overtime effort fails, then you should not agree to work overtime.

False Delivery

Of all the unprofessional behaviors that a programmer can indulge in, perhaps the worst of all is saying you are done when you know you aren't. Sometimes this is just an overt lie, and that's bad enough. But the far more insidious case is when we manage to rationalize a new definition of "done." We convince ourselves that we are done *enough*, and move on to the next task. We rationalize that any work that remains can be dealt with later when we have more time.

This is a contagious practice. If one programmer does it, others will see and follow suit. One of them will stretch the definition of "done" even more, and everyone else will adopt the new definition. I've seen this taken to horrible extremes. One of my clients actually defined "done" as "checked-in." The code didn't even have to compile. It's very easy to be "done" if nothing has to work!

When a team falls into this trap, managers hear that everything is going fine. All status reports show that everyone is on time. It's like blind men having a picnic on the railroad tracks: Nobody sees the freight train of unfinished work bearing down on them until it is too late.

Define "Done"

You avoid the problem of false delivery by creating an independent definition of "done." The best way to do this is to have your business analysts and testers create automated acceptance tests[5] that must pass before you can say that you are done. These tests should be written in a testing language such as FitNesse, Selenium, RobotFX, Cucumber, and so on. The tests should be understandable by the stakeholders and business people, and should be run frequently.

Help

Programming is *hard*. The younger you are the less you believe this. After all, it's just a bunch of `if` and `while` statements. But as you gain experience you begin to realize that the way you combine those `if` and `while` statements is critically

5. See Chapter 7, "Acceptance Testing."

important. You can't just slather them together and hope for the best. Rather, you have to carefully partition the system into small understandable units that have as little to do with each other as possible—and that's hard.

Programming is so hard, in fact, that it is beyond the capability of one person to do it well. No matter how skilled you are, you will certainly benefit from another programmer's thoughts and ideas.

Helping Others

Because of this, it is the responsibility of programmers to be available to help each other. It is a violation of professional ethics to sequester yourself in a cubicle or office and refuse the queries of others. Your work is not so important that you cannot lend some of your time to help others. Indeed, as a professional you are honor bound to offer that help whenever it is needed.

This doesn't mean that you don't need some alone time. Of course you do. But you have to be fair and polite about it. For example, you can let it be known that between the hours of 10 AM and noon you should not be bothered, but from 1 PM to 3 PM your door is open.

You should be conscious of the status of your teammates. If you see someone who appears to be in trouble, you should offer your help. You will likely be quite surprised at the profound effect your help can have. It's not that you are so much smarter than the other person, it's just that a fresh perspective can be a profound catalyst for solving problems.

When you help someone, sit down and write code together. Plan to spend the better part of an hour or more. It may take less than that, but you don't want to appear to be rushed. Resign yourself to the task and give it a solid effort. You will likely come away having learned more than you gave.

Being Helped

When someone offers to help you, be gracious about it. Accept the help gratefully and give yourself to that help. *Do not protect your turf.* Do not push the help away because you are under the gun. Give it thirty minutes or so. If by that time the person is not really helping all that much, then politely excuse yourself and terminate the session with thanks. Remember, just as you are honor bound to offer help, you are honor bound to accept help.

Learn how to *ask* for help. When you are stuck, or befuddled, or just can't wrap your mind around a problem, ask someone for help. If you are sitting in a team room, you can just sit back and say, "I need some help." Otherwise, use yammer, or twitter, or email, or the phone on your desk. Call for help. Again, this is a matter of professional ethics. It is unprofessional to remain stuck when help is easily accessible.

By this time you may be expecting me to burst into a chorus of *Kumbaya* while fuzzy bunnies leap onto the backs of unicorns and we all happily fly over rainbows of hope and change. No, not quite. You see, programmers *tend* to be arrogant, self-absorbed introverts. We didn't get into this business because we like *people.* Most of us got into programming because we prefer to deeply focus on sterile minutia, juggle lots of concepts simultaneously, and in general prove to ourselves that we have brains the size of a planet, all while not having to interact with the messy complexities of *other people.*

Yes, this is a stereotype. Yes, it is generalization with many exceptions. But the reality is that programmers do not tend to be collaborators.[6] And yet collaboration is critical to effective programming. Therefore, since for many of us collaboration is not an instinct, we require *disciplines* that drive us to collaborate.

MENTORING

I have a whole chapter on this topic later in the book. For now let me simply say that the training of less experienced programmers is the responsibility of those who have more experience. Training courses don't cut it. Books don't cut it. Nothing can bring a young software developer to high performance quicker

6. This is far more true of men than women. I had a wonderful conversation with @desi (Desi McAdam, founder of DevChix) about what motivates women programmers. I told her that when I got a program working, it was like slaying the great beast. She told me that for her and other women she had spoken to, the act of writing code was an act of nurturing creation.

than his own drive, and effective mentoring by his seniors. Therefore, once again, it is a matter of professional ethics for senior programmers to spend time taking younger programmers under their wing and mentoring them. By the same token, those younger programmers have a professional duty to seek out such mentoring from their seniors.

BIBLIOGRAPHY

[**Martin09**]**:** Robert C. Martin, *Clean Code*, Upper Saddle River, NJ: Prentice Hall, 2009.

[**Martin03**]**:** Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

Index