Clean Craftsmanship
Disciplines, Standards, and Ethics

Foreword by Stacia Heimgartner Viscardi, CST & Agile Mentor

Robert C. Martin
Praise for Clean Craftsmanship

“Bob’s Clean Craftsmanship has done a great job explaining the purposes of agile technical practices, along with a deep historical basis for how they came into existence, as well as positioning for why they will always be important. His involvement in history and formation of agility, thorough understanding of practices, and their purposes reflect vividly throughout the manuscript.”

—Tim Ottinger, well-known Agile Coach and author

“Bob’s writing style is excellent. It is easy to read and the concepts are explained in perfect detail for even a new programmer to follow. Bob even has some funny moments, which pleasantly snap you out of focus. The true value of the book is really in the cry for change, for something better . . . the cry for programmers to be professional . . . the realization that software is everywhere. Additionally, I believe there is a lot of value in all the history Bob provides. I enjoy that he doesn’t waste time laying blame for how we got to where we are now. Bob calls people to action, asking them to take responsibility by increasing their standards and level of professionalism, even if that means pushing back sometimes.”

—Heather Kanser

“As software developers, we have to continually solve important problems for our employers, customers, colleagues, and future selves. Getting the app to work, though difficult, is not enough, it does not make you a craftsman. With an app working, you have passed the app-titude test. You may have the aptitude to be a craftsman, but there is more to master. In these pages, Bob expresses clearly the techniques and responsibilities to go beyond the app-titude test and shows the way of the serious software craftsman.”

—James Grenning, author of Test-Driven Development for Embedded C and Agile Manifesto co-author

“Bob’s one of the very few famous developers with whom I’d like to work on a tech project. It’s not because he’s a good developer, famous, or a good communicator; it’s because Bob helps me be a better developer and a team member. He has spotted every major development trend, years ahead of others, and has been able to explain its importance, which encouraged me to learn. Back when I started—apart from being honest and a good person—the idea of craftsmanship and ethics was completely missing from this field. Now, it seems to be the most important thing professional developers can learn, even ahead of coding itself. I’m happy to see Bob leading the way again. I can’t wait to hear his perspective and incorporate it into my own practice.”

—Daniel Markham, Principal, Bedford Technology Group, Inc.
This page intentionally left blank
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. The series contains books that guide software professionals in the principles, patterns, and practices of programming, software project management, requirements gathering, design, analysis, testing, and others.

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

Make sure to connect with us! informit.com/socialconnect
Clean Craftsmanship

Disciplines, Standards, and Ethics

Robert C. Martin

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo
In memory of Mike Beedle
# Contents

- Foreword         
- Preface          
- Acknowledgments  
- About the Author 

## Chapter 1: Craftsmanship

## PART 1: The Disciplines

- Extreme Programming
  - The Circle of Life
- Test-Driven Development
- Refactoring
- Simple Design
- Collaborative Programming
- Acceptance Tests

## Chapter 2: Test-Driven Development

- Overview
- Software
Contents

Test Patterns 154
  Test-Specific Subclass 155
  Self-Shunt 156
  Humble Object 157
Test Design 160
  The Fragile Test Problem 160
  The One-to-One Correspondence 161
  Breaking the Correspondence 163
  The Video Store 164
  Specificity versus Generality 183
Transformation Priority Premise 184
  \{ \} \rightarrow \text{Nil} 186
  \text{Nil} \rightarrow \text{Constant} 187
  \text{Unconditional} \rightarrow \text{Selection} 188
  \text{Value} \rightarrow \text{List} 189
  \text{Statement} \rightarrow \text{Recursion} 189
  \text{Selection} \rightarrow \text{Iteration} 190
  \text{Value} \rightarrow \text{Mutated Value} 190
  \text{Example: Fibonacci} 191
  \text{The Transformation Priority Premise} 195
Conclusion 196

Chapter 5 Refactoring 197
What Is Refactoring? 199
The Basic Toolkit 200
  Rename 200
  Extract Method 201
  Extract Variable 202
  Extract Field 204
  Rubik's Cube 217
The Disciplines 217
  Tests 218
  Quick Tests 218
  Break Deep One-to-One Correspondences 218
  Refactor Continuously 219
  Refactor Mercilessly 219

xI
## Contents

Keep the Tests Passing! 219  
Leave Yourself an Out 220  
Conclusion 221

### Chapter 6 Simple Design 223  
YAGNI 226  
Covered by Tests 228  
  Coverage 230  
  An Asymptotic Goal 231  
  Design? 232  
  But There’s More 232  
Maximize Expression 233  
The Underlying Abstraction 235  
Tests: The Other Half of the Problem 236  
Minimize Duplication 237  
  Accidental Duplication 238  
Minimize Size 239  
  Simple Design 239

### Chapter 7 Collaborative Programming 241

### Chapter 8 Acceptance Tests 245  
The Discipline 248  
The Continuous Build 249

### Part II The Standards 251  
Your New CTO 252

### Chapter 9 Productivity 253  
We Will Never Ship S**T 254  
Inexpensive Adaptability 256  
We Will Always Be Ready 258  
Stable Productivity 259

### Chapter 10 Quality 261  
Continuous Improvement 262
Contents

Fearless Competence 263
Extreme Quality 264
We Will Not Dump on QA 265
  The QA Disease 266
QA Will Find Nothing 266
Test Automation 267
Automated Testing and User Interfaces 268
Testing the User Interface 269

Chapter 11  Courage 271
  We Cover for Each Other 272
  Honest Estimates 274
  You Must Say NO 276
  Continuous Aggressive Learning 277
  Mentoring 278

PART III  The Ethics 279
  The First Programmer 280
  Seventy-Five Years 281
  Nerds and Saviors 286
  Role Models and Villains 289
  We Rule the World 290
  Catastrophes 291
  The Oath 293

Chapter 12  Harm 295
  First, Do No Harm 296
    No Harm to Society 297
    Harm to Function 299
    No Harm to Structure 302
    Soft 303
    Tests 305
  Best Work 306
    Making It Right 307
    What Is Good Structure? 308
    Eisenhower’s Matrix 310
Programmers Are Stakeholders 312
Your Best 314
Repeatable Proof 316
  Dijkstra 316
  Proving Correctness 317
Structured Programming 319
Functional Decomposition 322
Test-Driven Development 323

Chapter 13 Integrity 327
  Small Cycles 328
    The History of Source Code Control 328
    Git 334
    Short Cycles 336
    Continuous Integration 337
    Branches versus Toggles 338
    Continuous Deployment 340
    Continuous Build 341
  Relentless Improvement 342
    Test Coverage 343
    Mutation Testing 344
    Semantic Stability 344
    Cleaning 345
    Creations 346
  Maintain High Productivity 346
    Viscosity 347
    Managing Distractions 349
    Time Management 352

Chapter 14 Teamwork 355
  Work as a Team 356
    Open/Virtual Office 356
  Estimate Honestly and Fairly 358
    Lies 359
  Honesty, Accuracy, Precision 360
  Story 1: Vectors 361
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Story 2: pCCU</td>
<td>363</td>
</tr>
<tr>
<td>The Lesson</td>
<td>365</td>
</tr>
<tr>
<td>Accuracy</td>
<td>365</td>
</tr>
<tr>
<td>Precision</td>
<td>367</td>
</tr>
<tr>
<td>Aggregation</td>
<td>369</td>
</tr>
<tr>
<td>Honesty</td>
<td>370</td>
</tr>
<tr>
<td>Respect</td>
<td>372</td>
</tr>
<tr>
<td>Never Stop Learning</td>
<td>373</td>
</tr>
</tbody>
</table>

Index                                                                 | 375  |
This page intentionally left blank
I remember meeting Uncle Bob in the spring of 2003, soon after Scrum was introduced to our company and technology teams. As a skeptical, fledgling ScrumMaster, I remember listening to Bob teach us about TDD and a little tool called FitNesse, and I remember thinking to myself, “Why would we ever write test cases that fail first? Doesn’t testing come after coding?” I often walked away scratching my head, as did many of my team members, and yet to this day I distinctly remember Bob’s palpable exuberance for code craftsmanship like it was only yesterday. I recall his directness one day as he was looking at our bug backlog and asking us why on earth we would make such poor decisions about software systems that we did not in fact own—“These systems are company assets, not your own personal assets.” His passion piqued our curiosity, and a year and half later, we had refactored our way to about 80 percent automated test coverage and a clean code base that made pivoting much easier, resulting in much happier customers—and happier teams. We moved lightning fast after that, wielding our definition of done like armor to protect us from the always-lurking code goblins; we had learned, in essence, how to protect us from ourselves. Over time, we developed a warmth for Uncle Bob, who came to truly feel like an uncle to us—a warm, determined, and courageous man who would over time help us learn to stand up for ourselves and do what was right. While some kids’ Uncle
Bobs taught them how to ride bicycles or fish, our Uncle Bob taught us to not compromise our integrity—and to this day, the ability and desire to show up to every situation with courage and curiosity has been the best lesson of my career.

I brought Bob’s early lessons with me on my journey as I ventured into the world as an agile coach and quickly observed for myself that the best product development teams figured out how to package up their own best practices for their unique contexts, for their particular customers, in their respective industries. I remembered Bob’s lessons when I observed that the best development tools in the world were only as good as their human operators—the teams who would figure out the best applications of those tools within their own domains. I observed that, sure, teams can reach high percentages of unit test coverage to check the box and meet the metric, only to find that a large percentage of those tests are flaky—metric was met, but value was not delivered. The best teams didn’t really need to care about metrics; they had purpose, discipline, pride, and responsibility—and the metrics in every case spoke for themselves. *Clean Craftsmanship* weaves all of these lessons and principles into practical code examples and experiences to illustrate the difference between writing something to meet a deadline versus actually building something sustainable for the future.

*Clean Craftsmanship* reminds us to never settle for less, to walk the Earth with fearless competence. This book, like an old friend, will remind you of what matters, what works, what doesn’t, what creates risk, and what diminishes it. These lessons are timeless. You may find that you already practice some of the techniques contained within, and I bet you’ll find something new, or at least something that you dropped because at some point you caved to deadlines or other pressures in your career. If you are new to the development world—whether in business or technology—you will learn from the very best, and even the most practiced and battle weary will find ways to improve themselves. Perhaps this book will help you find your passion again, renew your desire to improve your craft, or rededicate your energy to the search for perfection, regardless of the impediments on your horizon.
Software developers rule the world, and Uncle Bob is here again to remind us of the professional discipline of those with such power. He picks up where he left off with Clean Code; because software developers literally write the rules of humankind, Uncle Bob reminds us that we must practice a strict code of ethics, a responsibility to know what the code does, how people use it, and where it breaks. Software mistakes cost people their livelihoods—and their lives. Software influences the way we think, the decisions we make, and as a result of artificial intelligence and predictive analytics, it influences social and herd behavior. Therefore, we must be responsible and act with great care and empathy—the health and well-being of people depend on it. Uncle Bob helps us face this responsibility and become the professionals that our society expects, and demands, us to be.

As the Agile Manifesto nears its twentieth birthday at the writing of this foreword, this book is a perfect opportunity to go back to basics: a timely and humble reminder of the ever-increasing complexity of our programmatic world and how we owe it to the legacy of humankind—and to ourselves—to practice ethical development. Take your time reading Clean Craftsmanship. Let the principles seep into you. Practice them. Improve them. Mentor others. Keep this book on your go-to bookshelf. Let this book be your old friend—your Uncle Bob, your guide—as you make your way through this world with curiosity and courage.

—Stacia Heimgartner Viscardi, CST & Agile Mentor
This page intentionally left blank
Before we begin, there are two issues we need to deal with in order to ensure that you, my gentle reader, understand the frame of reference in which this book is presented.

**On the Term **Craftsmanship**

The beginning of the twenty-first century has been marked by some controversy over language. We in the software industry have seen our share of this controversy. One term that is often called out as a failure to be inclusive is *craftsman*.

I’ve given this issue quite a bit of thought and talked with many people of varying opinions, and I’ve come to the conclusion that there is no better term to use in the context of this book.

Alternatives to craftsman were considered, including craftsperson, craftsfolk, and crafter, among others. But none of those terms carries the historical gravitas of *craftsman*. And that historical gravitas is important to the message here.
Craftsman brings to mind a person who is deeply skilled and accomplished in a particular activity—someone who is comfortable with their tools and their trade, who takes pride in their work, and who can be trusted to behave with the dignity and professionalism of their calling.

It may be that some of you will disagree with my decision. I understand why that might be. I only hope you will not interpret it as an attempt to be exclusive in any way—for that is, by no means, my intent.

**On the One True Path**

As you read *Clean Craftsmanship: Disciplines, Standards, and Ethics*, you may get the feeling that this is the One True Path to Craftsmanship. It may be that for me, but not necessarily for you. I am offering this book to you as an example of my path. You will, of course, need to choose your own.

Will we eventually need One True Path? I don’t know. Perhaps. As you will read in these pages, the pressure for a strict definition of a software profession is mounting. We may be able to get away with several different paths, depending on the criticality of the software being created. But, as you will read in what follows, it may not be so easy to separate critical from noncritical software.

One thing I am certain of. The days of “Judges”¹ are over. It is no longer sufficient that every programmer does what is right in their own eyes. Some disciplines, standards, and ethics will come. The decision before us today is whether we programmers will define them for ourselves or have them forced upon us by those who don’t know us.

---

**INTRODUCTION TO THE BOOK**

This book is written for programmers and for managers of programmers. But in another sense, this book is written for all of human society. For it is we, programmers, who have inadvertently found ourselves at the very fulcrum of that society.

**FOR YOURSELF**

If you are a programmer of several years’ experience, you probably know the satisfaction of getting a system deployed and working. There is a certain pride you feel at having been part of such an accomplishment. You are proud of getting the system out the door.

But are you proud of the way you got that system out the door? Is your pride the pride of finishing? Or is your pride the pride of workmanship? Are you proud that the system has been deployed? Or are you proud of the way you built that system?

When you go home after a hard day of writing code, do you look at yourself in the mirror and say, “I did a good job today”? Or do you have to take a shower?

Too many of us feel dirty at the end of the day. Too many of us feel trapped into doing substandard work. Too many of us feel that low quality is expected and is necessary for high speed. Too many of us think that productivity and quality are inversely related.

In this book, I strive to break that mindset. This is a book about working well. This is a book about doing a good job. This is a book that describes the disciplines and practices that every programmer should know in order to work fast, be productive, and be proud of what they write every single day.
For Society

The twenty-first century marks the first time in human history that our society has become dependent, for its survival, on a technology that has acquired virtually no semblance of discipline or control. Software has invaded every facet of modern life, from brewing our morning coffee to providing our evening entertainment, from washing our clothes to driving our cars, from connecting us in a world-spanning network to dividing us socially and politically. There is literally no aspect of life in the modern world that is not dominated by software. And yet those of us who build this software are little more than a ragtag batch of tinkerers who barely have any idea what we are doing.

If we programmers had had a better grasp on what we do, would the 2020 Iowa Caucus results have been ready when promised? Would 346 people have died in the two 737 Max crashes? Would Knight Capital Group have lost $460 million in 45 minutes? Would 89 people have lost their lives in Toyota’s unintended acceleration accidents?

Every five years, the number of programmers in the world doubles. Those programmers are taught very little about their craft. They are shown the tools, given a few toy projects to develop, and are then tossed into an exponentially growing workforce to answer the exponentially growing demand for more and more software. Every day, the house of cards that we call software insinuates itself deeper and deeper into our infrastructure, our institutions, our governments, and our lives. And with every day, the risk of catastrophe grows.

Of what catastrophe do I speak? It is not the collapse of our civilization nor the sudden dissolution of all the software systems at once. The house of cards that is due to collapse is not composed of the software systems themselves. Rather, it is the fragile foundation of public trust that is at risk.

Too many more 737 Max incidents, Toyota unintended acceleration incidents, Volkswagen California EPA incidents, or Iowa Caucus incidents—too many
more cases of high-profile software failures or malfeasance—and the lack of our discipline, ethics, and standards will become the focus of a distrustful and enraged public. And then the regulations will follow: regulations that none of us should desire; regulations that will cripple our ability to freely explore and expand the craft of software development; regulations that will put severe restrictions on the growth of our technology and economy.

It is not the goal of this book to stop the headlong rush into ever-more software adoption. Nor is it the goal to slow the rate of software production. Such goals would be a waste of effort. Our society needs software, and it will get it no matter what. Attempting to throttle that need will not stop the looming catastrophe of public trust.

Rather, it is the goal of this book to impress upon software developers and their managers the need for discipline and to teach those developers and managers the disciplines, standards, and ethics that are most effective at maximizing their ability to produce robust, fault-tolerant, effective software. It is only by changing the way that we programmers work, by upping our discipline, ethics, and standards, that the house of cards can be shored up and prevented from collapse.

**The Structure of This Book**

This book is written in three parts that describe three levels: disciplines, standards, and ethics.

*Disciplines* are the lowest level. This part of the book is pragmatic, technical, and prescriptive. Programmers of all stripes will benefit from reading and understanding this part. Within the pages of this part are several references to videos. These videos show the rhythm of the test-driven development and refactoring disciplines in real time. The written pages of the book try to capture that rhythm as well, but nothing serves quite so well as videos for that purpose.
Standards are the middle level. This section outlines the expectations that the world has of our profession. This is a good section for managers to read, so that they know what to expect of professional programmers.

Ethics are at the highest level. This section describes the ethical context of the profession of programming. It does so in the form of an oath, or a set of promises. It is laced with a great deal of historical and philosophical discussion. It should be read by programmers and managers alike.

A Note for Managers

These pages contain a great deal of information that you will find beneficial. They also contain quite a bit of technical information that you probably don’t need. My advice is that you read the introduction of each chapter and stop reading when the content becomes more technical than you need. Then go to the next chapter and start again.

Make sure you read Part II, “The Standards,” and Part III, “The Ethics.” Make sure you read the introductions to each of the five disciplines.

Register your copy of Clean Craftsmanship on the InformIT site for convenient access to the companion videos, along with updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780136915713) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access the companion videos. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.
Acknowledgments

Thank you to my intrepid reviewers: Damon Poole, Eric Crichlow, Heather Kanser, Tim Ottinger, Jeff Langr, and Stacia Viscardi. They saved me from many a faltering step.

Thank you also to Julie Phifer, Chris Zahn, Menka Mehta, Carol Lallier, and to all those at Pearson who work so tirelessly to make these books work out so well.

As always, thank you to my creative and talented illustrator, Jennifer Kohnke. Her pictures always make me smile.

And, of course, thank you to my lovely wife and wonderful family.
This page intentionally left blank
Robert C. Martin (Uncle Bob) wrote his first line of code at the age of 12 in 1964. He has been employed as a programmer since 1970. He is cofounder of cleancoders.com, offering online video training for software developers, and founder of Uncle Bob Consulting LLC, offering software consulting, training, and skill development services to major corporations worldwide. He served as the Master Craftsman at 8th Light, Inc., a Chicago-based software consulting firm.
Mr. Martin has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He is also the creator of the acclaimed educational video series at cleancoders.com.

Mr. Martin has authored and edited many books, including the following:

* 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
* The Clean Coder
* Clean Architecture: A Craftsman’s Guide to Software Structure and Design
* Clean Agile: Back to Basics

A leader in the industry of software development, Mr. Martin served three years as the editor-in-chief of the C++ Report, and he served as the first chairperson of the Agile Alliance.
The dream of flying is almost certainly as old as humanity. The ancient Greek myth describing the flight of Daedalus and Icarus dates from around 1550 BCE. In the millennia that followed, a number of brave, if foolish, souls have strapped ungainly contraptions to their bodies and leapt off cliffs and towers to their doom in pursuit of that dream.

Things began to change about five hundred years ago when Leonardo DaVinci drew sketches of machines that, though not truly capable of flight, showed some reasoned thought. It was DaVinci who realized that flight could be possible because air resistance works in both directions. The resistance caused by pushing down on the air creates lift of the same amount. This is the mechanism by which all modern airplanes fly.

DaVinci’s ideas were lost until the middle of the eighteenth century. But then began an almost frantic exploration into the possibility of flight. The eighteenth and nineteenth centuries were a time of intense aeronautical research and experimentation. Unpowered prototypes were built, tried, discarded, and improved. The science of aeronautics began to take shape. The forces of lift, drag, thrust, and gravity were identified and understood. Some brave souls made the attempt.

And some crashed and died.

In the closing years of the eighteenth century, and for the half century that followed, Sir George Cayley, the father of modern aerodynamics, built experimental rigs, prototypes, and full-sized models culminating in the first manned flight of a glider.

And some still crashed and died.

Then came the age of steam and the possibility of powered manned flight. Dozens of prototypes and experiments were performed. Scientists and enthusiasts alike joined the gaggle of people exploring the potential of flight. In 1890, Clément Ader flew a twin-engine steam-powered machine for 50 meters.
And some still crashed and died.

But the internal combustion engine was the real game-changer. In all likelihood, the first powered and controlled manned flight took place in 1901 by Gustave Whitehead. But it was the Wright Brothers who, on December 17, 1903, at Kill Devil Hills, North Carolina, conducted the first truly sustained, powered, and controlled manned flight of a heavier-than-air machine.

And some still crashed and died.

But the world changed overnight. Eleven years later, in 1914, biplanes were dogfighting in the air over Europe.

And though many crashed and died under enemy fire, a similar number crashed and died just learning to fly. The principles of flight might have been mastered, but the technique of flight was barely understood.

Another two decades, and the truly terrible fighters and bombers of World War II were wreaking havoc over France and Germany. They flew at extreme altitudes. They bristled with guns. They carried devastating destructive power.

During the war, 65,000 American aircraft were lost. But only 23,000 of those were lost in combat. The pilots flew and died in battle. But more often they flew and died when no one was shooting. We still didn’t know how to fly.

Another decade saw jet-powered craft, the breaking of the sound barrier, and the explosion of commercial airlines and civilian air travel. It was the beginning of the jet age, when people of means (the so-called jet set) could leap from city to city and country to country in a matter of hours.

And the jet airliners tore themselves to shreds and fell out of the sky in terrifying numbers. There was so much we still didn’t understand about making and flying aircraft.
That brings us to the 1950s. Boeing 707s would be flying passengers from here to there across the world by the end of the decade. Two more decades would see the first wide-body jumbo jet, the 747.

Aeronautics and air travel settled down to become the safest and most efficient means of travel in the history of the world. It took a long time, and cost many lives, but we had finally learned how to safely build and fly airplanes.\(^1\)

Chesley Sullenberger was born in 1951 in Denison, Texas. He was a child of the jet age. He learned to fly at age sixteen and eventually flew F4 Phantoms for the Air Force. He became a pilot for US Airways in 1980.

On January 15, 2009, just after departure from LaGuardia, his Airbus A320 carrying 155 souls struck a flock of geese and lost both jet engines. Captain Sullenberger, relying on over twenty thousand hours of experience in the air, guided his disabled craft to a “water landing” in the Hudson River and, through sheer indomitable skill, saved every one of those 155 souls. Captain Sullenberger excelled at his craft. Captain Sullenberger was a craftsman.

The dream of fast, reliable computation and data management is almost certainly as old as humanity. Counting on fingers, sticks, and beads dates back thousands of years. People were building and using abacuses over four thousand years ago. Mechanical devices were used to predict the movement of stars and planets some two thousand years ago. Slide rules were invented about four hundred years ago.

In the early nineteenth century, Charles Babbage started building crank-powered calculating machines. These were true digital computers with memory and arithmetic processing. But they were difficult to build with the metalworking technology of the day, and though he built a few prototypes, they were not a commercial success.

\(^1\) The 737 Max notwithstanding.
In the mid-1800s, Babbage attempted to build a much more powerful machine. It would have been steam powered and capable of executing true programs. He dubbed it the *Analytical Engine*.

Lord Byron’s daughter, Ada—the Countess of Lovelace—translated the notes of a lecture given by Babbage and discerned a fact that had apparently not occurred to anyone else at the time: *the numbers in a computer need not represent numbers at all but can represent things in the real world*. For that insight, she is often called the world’s first true programmer.

Problems of precise metalworking continued to frustrate Babbage, and in the end, his project failed. No further progress was made on digital computers throughout the rest of the nineteenth and early twentieth centuries. During that time, however, mechanical *analog* computers reached their heyday.

In 1936, Alan Turing showed that there is no general way to prove that a given Diophantine\(^2\) equation has solutions. He constructed this proof by imagining a simple, if infinite, digital computer and then proving that there were numbers that this computer could not calculate. As a consequence of this proof, he invented finite state machines, machine language, symbolic language, macros, and primitive subroutines. He invented, what we would call today, software.

At almost exactly the same time, Alonzo Church constructed a completely different proof for the same problem and consequently developed the lambda calculus—the core concept of functional programming.

In 1941, Konrad Zuse built the first electromechanical programmable digital computer, the Z3. It consisted of more than 2,000 relays and operated at a clock rate of 5 to 10Hz. The machine used binary arithmetic organized into 22-bit words.

During World War II, Turing was recruited to help the “boffins” at Bletchley Park decrypt the German Enigma codes. The Enigma machine was a simple

---

\(^2\) Equations of integers.
digital computer that randomized the characters of textual messages that were typically broadcast using radio telegraphs. Turing aided in the construction of an electromechanical digital search engine to find the keys to those codes.

After the war, Turing was instrumental in building and programming one of the world’s first electronic vacuum tube computers—the Automatic Computing Engine, or ACE. The original prototype used 1,000 vacuum tubes and manipulated binary numbers at a speed of a million bits per second.

In 1947, after writing some programs for this machine and researching its capabilities, Turing gave a lecture in which he made these prescient statements:

> We shall need a great number of mathematicians of ability [to put the problems] into a form for computation.

> One of our difficulties will be the maintenance of an appropriate discipline, so that we do not lose track of what we are doing.

And the world changed overnight.

Within a few years, core memory had been developed. The possibility of having hundreds of thousands, if not millions, of bits of memory accessible within microseconds became a reality. At the same time, mass production of vacuum tubes made computers cheaper and more reliable. Limited mass production was becoming a reality. By 1960, IBM had sold 140 model 70x computers. These were huge vacuum tube machines worth millions of dollars.

Turing had programmed his machine in binary, but everyone understood that was impractical. In 1949, Grace Hopper had coined the word compiler and by 1952 had created the first one: A-0. In late 1953, John Bachus submitted the first FORTRAN specification. ALGOL and LISP followed by 1958.

The first working transistor was created by John Bardeen, Walter Brattain, and William Shockley in 1947. They made their way into computers in 1953. Replacing vacuum tubes with transistors changed the game entirely. Computers became smaller, faster, cheaper, and much more reliable.
By 1965, IBM had produced 10,000 model 1401 computers. They rented for $2,500 per month. This was well within the reach of midsized businesses. Those businesses needed programmers, and so the demand for programmers began to accelerate.

Who was programming all these machines? There were no university courses. Nobody went to school to learn to program in 1965. These programmers were drawn from business. They were mature folks who had worked in their businesses for some time. They were in their 30s, 40s, and 50s.

By 1966, IBM was producing 1,000 model 360 computers every month. Businesses could not get enough. These machines had memory sizes that reached 64kB and more. They could execute hundreds of thousands of instructions per second.

That same year, working on a Univac 1107 at the Norwegian Computer Center, Ole-Johan Dahl and Kristen Nygard invented Simula 67, an extension of ALGOL. It was the first object-oriented language.

Alan Turing’s lecture was only two decades in the past!

Two years later, in March 1968, Edsger Dijkstra wrote his famous letter to the *Communications of the ACM (CACM)*. The editor gave that letter the title “Go To Statement Considered Harmful.”

Structured programming was born.

In 1972, at Bell Labs in New Jersey, Ken Thompson and Dennis Ritchie were between projects. They begged time on a PDP 7 from a different project team and invented UNIX and C.

Now the pace picked up to near breakneck speeds. I’m going to give you a few key dates. For each one, ask yourself how many computers are in the

---

world? How many programmers are in the world? And where did those programmers come from?

1970—Digital Equipment Corporation had produced 50,000 PDP-8 computers since 1965.
1970—Winston Royce wrote the “waterfall” paper, “Managing the Development of Large Software Systems.”
1971—Intel released the 4004 single-chip microcomputer.
1974—Intel released the 8080 single-chip microcomputer.
1977—Apple released the Apple II.
1979—Motorola released the 68000, a 16-bit single-chip microcomputer.
1980—Bjarne Stroustrup invented C with Classes (a preprocessor that makes C look like Simula).
1980—Alan Kay invented Smalltalk.
1981—IBM released the IBM PC.
1983—Apple released the 128K Macintosh.
1983—Stroustrup renamed C with Classes to C++.
1986—Stroustrup published The C++ Programming Language (Addison-Wesley).
1991—Grady Booch published Object-Oriented Design with Applications (Benjamin/Cummings).
1991—James Gosling invented Java (called Oak at the time).
1995—Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley) was written by Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson.
1995—Yukihiro Matsumoto released Ruby.
1996—Sun Microsystems released Java.
1999—Microsoft invented C#/.NET (then called Cool).
2001—The Agile Manifesto was written.

Between 1970 and 2000, the clock rates of computers increased by three orders of magnitude. Density increased by four orders of magnitude. Disk space increased by six or seven orders of magnitude. RAM capacity increased by six or seven orders of magnitude. Costs had fallen from dollars per bit to dollars per gigabit. The change in the hardware is hard to visualize, but just summing up all the orders of magnitude I mentioned leads us to about a thirty orders of magnitude increase in capability.

And all this in just over fifty years since Alan Turing’s lecture.

How many programmers are there now? How many lines of code have been written? How good is all that code?

Compare this timeline with the aeronautical timeline. Do you see the similarity? Do you see the gradual increase in theory, the rush and failure of the enthusiasts, the gradual increase in competence? The decades of not knowing what we were doing?

And now, with our society depending, for its very existence, on our skills, do we have the Sullenbergers whom our society needs? Have we groomed the programmers who understand their craft as deeply as today’s airline pilots understand theirs? Do we have the craftsmen whom we shall certainly require?

Craftsmanship is the state of knowing how to do something well and is the outcome of good tutelage and lots of experience. Until recently, the software industry had far too little of either. Programmers tended not to remain programmers for long, because they viewed programming as a steppingstone into management. This meant that there were few programmers who acquired enough experience to teach the craft to others. To make matters worse, the number of new programmers entering the field doubles every five years or so, keeping the ratio of experienced programmers far too low.
The result has been that most programmers never learn the disciplines, standards, and ethics that could define their craft. During their relatively brief career of writing code, they remain unapprenticed novices. And this, of course, means that much of the code produced by those inexperienced programmers is substandard, ill structured, insecure, buggy, and generally a mess.

In this book, I describe the standards, disciplines, and ethics that I believe every programmer should know and follow in order to gradually acquire the knowledge and skill that their craft truly requires.
INDEX

A
Abstraction
clarifying levels of, 201
Dijkstra’s development of, 317–319, 320
polymorphism for, 225
stepdown rule for, 202
underlying, 235–236
Acceptance tests, 14, 18, 246–249
Accidental duplication, 237–238
Accounting, 20–22, 23
Accurate estimates, 360–361, 365–367
Ada, Countess of Lovelace, 5, 280.
See also N345TS
Adaptability, inexpensive, 256–257
Ader, Clément, 2
Aeronautics, 1–4
Aggregate estimates, 369–370
Agile methods
readiness via, 258, 259
story points, 275
test automation via, 226
XP as, 14
ALGOL, 6, 7, 282

Algorithms
bowling game scoring, 62–79
bubble sorting, 82–87
to find prime factors, 52–62
incremental derivation of, 61, 82, 86
integer stacking, 36–52
proving correctness of, 316–323
quick sorting, 94–95
sine calculation, 127–138
text wrapping, 95–103
video store rentals, 164–183
API function, 269
Architecture, 143–145
Arrange/Act/Assert test pattern,
103–104, 247, 267
Authenticator interface, 111
Automated tests, 246, 267–269, 340
Automatic Computing Engine (ACE), 6, 280

B
Babbage, Charles, 4–5, 280
Backus, John, 6, 282, 333
Balance, 20
Beck, Kent
  design principles of, 226, 239
  Extreme Programming, 14
  on making it right, 306
  on refactoring, 219
  on simple design, 224, 228
Behavior
  anticipating, 300
  refactoring to preserve, 199
  vs. structure, 72, 303–304, 307, 311
Behavior-Driven Development (BDD),
  104–105, 107
Bergensten, Jens, 289
Best work, 306–315
Boeing, 4
Boundaries, 143–144
Bowling score computation, 62–79,
  104, 203
Branches, toggles vs., 338–340
Bubble sort algorithms, 87, 164
Bugs. See also Debugging
  accumulation of, 314
  as found by QA, 266, 267
  quick patches for, 302
  quick tests for, 218
  as unacceptable, 264
Builds
  continuous, 249, 341–342
  speeding up, 347–348
Business analysis (BA) teams, 246, 248
Business rules
  decoupling databases from, 148
  isolated from tests, 268–269
  testing the, 150
  user interface vs., 141, 142

C
C language, 284
Cascading Style Sheets (CSS) code,
  255–256
Catastrophes, 291–293
Cayley, Sir George, 2
CCU-CMU, 364
Certainty
  cost of, 358, 370
  fragile tests and, 139, 140
  London school of, 141
Changes
  fear of making, 31–32, 219, 263
  inexpensive, 256–257, 262
  keeping up with, 373–374
  learning to prepare for, 277
  as software necessity, 303–304, 307
  to user interface, 268
  via git, 335
  via good structure, 308–310
Chicago school of TDD, 142
Chief Technical Officer (CTO), 252
Church, Alonzo, 5, 62
Circle of Life, 14–15
Clean Agile (Martin), 358
Clean Architecture (Martin), 143, 145
Clean Code (Martin), 198
Clean Craftsmanship (Martin), xxv–xxvi
COBOL, 282, 283
Code
  building blocks of, 321
  cleaning of, 32–34, 306, 342, 345
  decoupled, 30–31, 232
  decoupling tests from, 161–184,
    218, 232
  designing testable, 30
  do no harm to, 302–303
  early example of, 233
  expressive, 233–235
  fear of broken, 31–32, 219
  fragile tests and, 160–161
  by the inexperienced, 9–10
  proving correctness of, 316–323
  structural vs behavioral changes, 72
  test and code communication, 236
  test coverage of, 230
  two schools of TDD, 140–142
Coding
  history of, 280–286
specific tests for generic code, 59, 82, 183–184
test suite holes, 28–29
and three laws of TDD, 23–26
transformations, 185
Collaborating components, 144–145
Collaborative programming
discipline of, 242–244
improvement via, 356
remotely, 357
to replace yourself, 272, 273
as XP practice, 14, 17–18
COLT (Central Office Line Tester), 361
Commitments, 368–369, 370–372
Competence, fearless, 263–264
Compilation tests, 36–52
Complexity, incremental, 92
Components, 144–145
Computers
history of, 4–9, 280–286
in pop culture, 287
Concurrent Versions System (CVS), 333, 334, 336
Constant to variable transformation,
187–188
Continuous build, 249, 341–342
Continuous integration, 337
Continuous learning, 277–278, 373–374
Controllers
in component hierarchy, 144–145
and GUI input tests, 153–154
in Humble Object test pattern, 159
testing, 153
Countess. See Ada
Coupling. See also Decoupling
fragile test problem and, 139
high- and low-level details, 152, 218, 224–225
minimizing, 184
one-to-one, 161–162
Courage, 271–278
Coverage, test, 229–231, 239, 343–344
Craftsmanship
in the aeronautics industry, 4
defined, xxii–xxii
as developer responsibility, 303
foundation disciplines of, 14–18, 293
history of computer, 4–9
by programmers, 9–10
provable code as, 324–325
Craig, Philip, 108
D
Dahl, Ole-Johann, 283
Database testing, 148–150
DaVinci, Leonardo, 2
Debugging
avoiding, 199
commitment to total, 264, 266
minimizing, 26–27
by QA, 265–267
speeding up, 349
Decidability problem, 62
Decision matrix, Eisenhower’s,
310–312
Decision problem, 280
Decomposition, functional, 322–323
Decoupling
code, for testability, 232
databases, 148
high- and low-level details, 236, 308
importance of, 224–225
production code/test, 161–184, 218
Degenerate tests
solutions via, 83–84, 98–99
staircase tests as, 66
as TDD rule, 53–54
Dependencies
dependency inversion principle, 143
eliminating unnecessary,
112–113, 114
management of, via SOLID, 309–310
with production data, 116
Dependency Rule, 144–145
Deployment
and best work, 307
continuous, 340–341
readiness for, 258
speeding up, 349
for startups, 304–305
and structural changes, 308
Derivation, incremental. See Incremental derivation
Design
changes to, 232
four principles of, 239
fragility as problematic, 160–161
high test coverage, 230–232
holes, 28–29
minimal size in, 239
outside-in vs. inside-out, 141–142
prompt fixing of flaws, 70–71
simple, 17, 224–226, 228–229
smells, 308, 309
test design, 160–184
YAGNI, 226–228
Development, test-driven. See Test-driven development (TDD)
Digital Equipment Corporation, 284
Digital switching, 364
Dijkstra, Edsger, 7, 282, 316–323
Diophantine equation, 62
Disciplines
acceptance tests, 18, 248–249
Alan Turing on, 281
in Clean Craftsmanship, xxv
collaborative programming, 17–18
failing at, 255, 256
focus and, 13
refactoring, 16, 217–221
self- vs. other-imposed, xxv
simple design, 17
test-driven development (TDD), 15–16
YAGNI, 227
Distractions, managing, 349–354
Double-entry bookkeeping, 20–22
Dummies, 111–114
Duplication
accidental, 238
duplicate tests, 66
minimizing code, 237–238
E
Eisenhower’s decision matrix, 310–312
Elements variable, 47, 51
Emotional stress, 351
“Endo-Testing: Unit Testing with Mock Objects” (Freeman, McKinnon, Craig), 108
Engineering practices, 14
Enigma machine, 5
Entangled software, 224
Enumeration, 317, 321
Errors, 70, 71
Estimates, honest and fair, 274–275, 358–372
Ethics
in Clean Craftsmanship, xxvi
oath of, 293
single vs. multiple, xxii
Expectations, 252
Explanatory variables, 204
Exploratory testing, 267
Expressivity
code language for, 233–234
as design principle, 239
and real vs. accidental duplicates, 238
role of code and tests in, 236
External services, 108, 110
Extract Field refactoring, 204–216
Extract Method refactoring, 68, 201–202, 208
Extract Superclass refactoring, 214
Extract Variable refactoring, 202–204
Extreme Programming Explained (Beck), 228
Extreme Programming (XP), 13–14, 219, 226, 258
## Index

**F**
- **Factors** variable, 54–55
- Failure tests
  - algorithms via, 82
  - input/output (IO), 108
  - for integer sort algorithm, 83
  - in integer stack example, 36–52
  - multiple solutions to, 88
  - in prime factors example, 52–62
- Fakes, 124–126
- Fear
  - of broken code, 31–32, 219
  - fearless competence, 263–264
- Fibonacci kata, 191–194
- Finite state machines, 105–107
- First-in-last-out (FILO) behavior test, 50, 52
- FitNesse, 247, 337
- Flaws, design, 70–71
- Flexibility, 141, 142, 345
- Flight, history of, 1–4
- Flow, the, 352
- Fortran, 282, 319–320
- Fowler, Martin
  - Chicago school of TDD, 142
  - on refactoring, 164, 198, 221
  - on simple design, 229
- Fragile tests
  - certainty and, 139, 140
  - as design problem, 160–161
  - mocks and, 139–140
  - refactoring to prevent, 218
  - test specificity to prevent, 59, 232
  - user interface links and, 268
- Fragility, software, 308
- Freeman, Steve, 108, 141
- Fun, 30
- Functional decomposition, 322–323
- Functional programming, 62, 196
- Functions
  - extract refactoring of, 201
  - function to nil transformation, 186–187
  - harm to, 299–301
  - minimizing size of, 239
  - misplaced responsibility of, 71
  - replacing comments with, 75
  - testing query, 148–150
  - tests as constraining, 127
  - tests tied to implementation of, 120
  - uncertainty principle in, 132
  - and underlying abstraction, 235
  - vs. cleaning code, 306

**G**
- Gateway interface, 149
- Generalization, 59, 183–184
- Generic code, 59
- Git, 334–336
- Git reset —hard, 220
- Given-When-Then (GWT), 104, 105, 248
- Goethe, Johann Wolfgang von, 21
- GOTO statements, 320, 321, 322
- Green step
  - as transformative, 185
  - in video rental algorithm, 165, 166, 167, 169, 173
- GUI (graphic user interface)
  - inputs, 153–154
  - testing, 150–153

**H**
- Haines, Corey, 229
- Handwashing, 12–13
- Harm, do no
  - to function, 299–301
  - to society, 296–297
  - to structure, 302–303
  - via easy changeability, 303–304
  - via tests, 305–306
- HealthCare.gov, 297–298
- Hendrickson, Chet, 224
- History of flight, 1–4
- Hooks, 226, 227–228
Index

Hopper, Grace, 6, 282
Horn, Michael, 289
Humble Object pattern, 157–160

I
IBM, 6, 7, 282, 283
If statements, 58, 60, 202
Immobility, 309
Implement Interface command, 114
Implementation, function, 120
Improvement, continuous, 262, 342–346
Incremental derivation
as solution to getting stuck, 95, 98
solving failing tests as, 86
TDD as technique for, 61, 82
writing tests via, 103
Induction, 317, 321
Inflexible software, 256–257
Inside-out design, 142
Integration tests, 182–183
Integrity
via high productivity, 346–354
via relentless improvement, 342–346
via small cycles, 328–342
Interactors
in component hierarchy, 144–145
GUI, 152, 154
Iteration, 321

J
JBehave, 104
Jennings, Jean, 282
JUnit Recipes (Rainsberger and Stirling), 154

K
Keogh, Liz, 105
Knight Capital Group, 299–300, 302
Koss, Bob, 62–79

L
Languages
continuous learning of, 277, 373–374
expressive code, 233–234
first programming, 7, 282–284
Law of trichotomy, 89
Learning
continuous, 277–278, 373–374
from mentors, 278, 286
via cleaning code, 345
Lies, estimates as, 359
Login test, 348
London school of TDD, 141
Loops, unwound, 57
Lovelace, Countess of, 5. See also Ada

M
MacKinnon, Tim, 108, 120
Managers
commitments to, 370–372
emotional confrontation by, 276
information for, xxvi
perspective of, 252
Manual testing, 267, 270
Marick, Brian, 305
Matz, Chris, 104
McCarthy, John, 282
Mean and sigma, 365–367
Meetings, 330
Memory, core, 6
Mentoring, 278, 286
Merges, 332, 334, 336, 337
Meszaros, Gerard, 110, 154
Minimizing size, 237, 239
Misplaced responsibility, 71
Mob programming, 242, 244, 357
Mock objects
defined, 109
fragility of testing with, 139
in test doubles, 108–110, 121–123
when to use, 126
Mojang, 289
Mood, 351
Music, 350–351
Mutation testing, 344
N
Nil to constant transformation, 187
“No,” saying, 276, 371
North, Dan, 104
N345TS. See Ada
Nygard, Kristen, 283

O
Object-oriented models, 64
One-to-one correspondences, 161–162, 218
Open offices, 356–358
Optimistic locking, 333
Outside-in design, 141

P
Pair programming, 242, 244, 352, 357
Parent-child algorithms, 299
Patterns, test, 155–160
Perspective, 252
Pessimistic locking, 334
Polymorphism, 110, 225
Pomodoro technique, 242, 352–354
Power Peg code, 299, 302
Precise estimates, 360–361, 367–369
Presenters
  in component hierarchy, 144–145
defined, 151
GUI, 151–152
  in Humble Object test pattern,
  157–159
Pride, 33–34
Prime factors algorithm, 52–62, 185
Probability distributions, 365
Procedural Programming, 62
Production data, 116
Production releases, 340
Productivity
  high, 346–354
  inexpensive adaptability in, 256–257
  perpetual readiness in, 258–259
  quality standard for, 254–256
  stable, 259–260
Program Evaluation and Review Technique (PERT), 369–370
Programmer tests, 182
Programmers
  continuous improvement by, 262
  as craftsmen, xxii
  demographics of, 284–286
  Edsger Dijkstra, 316–323
  as heroes and villains, 289–290
  history of, 5–9, 280–286
  inexperience of, 9–10
  as nerds and saviors, 287–288
  one path for, xxii
  perspective of, 252
  pop culture depictions of, 287–288
  responsibility of. See Responsibility
  senior, 297
  senior/junior partnerships, 244
  as stakeholders, 312–313
  teaching new, 278
  undisciplined, 256
Programming
  collaborative, 17–18, 242–244, 356
  decidability problem in, 62
  as finite state transitions, 107
  functional, 196
  history of, 280–286
  London school of, 141
  modular, 330–332
  procedural and functional, 62
  structured, 319–322
Proof, repeatable, 316
Property testing, 140
Pryce, Nat, 141
Public class stack, 36–52
Public trust, xxiv–xxv
Push function, 41

Q
Quality
  continuously improved, 262
definition of poor, 255
extreme, 264
fearless competence on, 263–264
readiness for QA review, 265–267
test automation, 267–269
testing the user interface, 269–270
Quality assurance (QA) department
acceptance test readability by, 246
readiness for review by, 265–267
tests by, 248
Queries, testing, 148–150
Quick sort algorithm, 94–95
Quick tests, 218, 324

R
Rainsberger, J.B., 154
Readiness, 258–259
RealAuthenticator, 111–112, 116
Recursion, 100
Red/Green/Refactor loop
cleanup via, 39
as continuous, 219
as fourth TDD law, 34–35
as functional decomposition, 323
moral foundation for, 312
as refactor cycle, 199
transformative priorities and, 185
Red step
as additive, 185
in video rental algorithm, 165, 166, 168–169, 172
Refactor step
as restorative, 185
in video rental algorithm, 165–166
Refactoring
basic toolkit for, 200–217
basis of, 232–233
cleaning as, 345
code generalization via, 184
consistent, 219
defined, 72, 199
disciplines of, 16, 217–221
as fourth TDD law, 34–35
and fragile tests, 139
IDE’s Extract Method, 68
Red/Green/Refactor loop, 34–35, 39, 185
restorative, 185
as Rubik’s cube solutions, 215
in video rental algorithm, 164–183
vs. placing hooks, 227
as XP practice, 14
Refactoring (Fowler), 198, 221
Remote work, 357
Rename refactoring, 200–201
Respect, 372–373
Responsibility
and collaborative programming, 243
of continuous learning, 374
to do no harm, 296–303
misplaced, 71
of programmers, 290–293, xxiv
recognition of, 306
to replace yourself, 273
single responsibility principle, 201, 214
of software developers, xxiv
as a stakeholder, 312–313
Revision Control System, 333
Rigidity, 308
Ritchie, Dennis, 284
Rochkind, Marc, 333
Role Models, 289
RSpec, 104
Rubik’s cube, 215

S
Science, software, 323–324
Scrum, 258
Selection, 321
Selection to iteration transformation, 190
Self-Shunt pattern, 156
Semantic stability, 344–345
Semmelweis, Ignaz, 13
Sequence, 321
Setup method, 103
Short cycles, 334–336
Sieve of Eratosthenes, 53, 61
Sigma, 366, 369
Simplicity
  design, 14, 17, 224–226
  fundamental aspect of, 235
  incremental derivation, 61, 82, 86
  rule of test, 95–96, 103
  via expressive language, 233–234
  via technology advances, 227
Simula 67, 7, 283
Sine calculation, 127–138
Single responsibility principle, 201, 214, 239, 246
Small cycles, 328–342
Smalltalk, 108
Society
  dependence on software, 9, 291, 306
  harm to, 297–298
  responsibility to, 292
  views on programmers, 286–288
Software
  catastrophes, 291–293
  changes to, 224, 226, 303–304
  failed estimates on, 358
  GUI, 150–151
  invention of, 5
  messy, harmful, 302
  pervasiveness of, 290–291
  proofs, 316–323
  public trust in, xxiv–xxv
  rigid, fragile, immobile, 256, 308–309, 345
  as a science, 323–324
  similarity to accounting, 23
  tests as documentation of, 26–28
  untangled, 224
SOLID principles, 17, 143, 309–310
Sorting algorithms
  bubble sort, 82–87, 95
  quick sort, 94–95
Source code
  continuous build of, 249
  control, 328–334
  dependencies, 143, 153, 225
  editors, 237
  harm to, 303
Source Code Control System (SCCS), 333
Specificity, 183–184, 202
Spies
  fragility of testing with, 138, 139
  mocks as, 121
  in sine calculation algorithm, 133–135
  in test doubles, 118–120
  for testing across boundaries, 144
Stabilization sprints, 258
Stack of integers, writing a, 36–52
StackTest, 37
Stairstep tests, 66
Stakeholders, 312–313
Standards
  in Clean Craftsmanship, xxvi
  courage, 271–278
  defined, 252
  differing perspectives on, 252
  productivity, 253–260
  provable code, 324–325
  quality, 261–270
  responsibility and, xix
  single vs. multiple, xxii
Startups, 304
State/Transition diagram, 105–106
Statement to recursion transformation, 189–190
Stepdown rule, 202
Stevenson, Chris, 104
Stirling, Scott, 154
Story points, 275
Structure
  behavior vs., 303–304, 307, 311
  and collaborative programming, 244
  duplication and, 237
good, 308–310
harm to, 302–303
Structured programming, 7, 319–323
Stubs
  and business rule isolation, 269–270
  spies as, 118–119
  in test double process, 115–117
Subversion, 334
Subway turnstiles, 105–106
Sullenberger, Chesley, 4
Surviving mutations, 344

T
Taylor series, 128, 130, 132–133
Teamwork
  collaborative programming, 242–244
  continuous integration and, 337
  covering for each other, 272
  honest/fair estimates as, 358–372
  knowledge sharing via, 356
  open/virtual offices, 356–358
  respect, 372–373
Technological advances, 227–228
Teradyne, 361
Test doubles
  boundary crossing and, 144
  dummies, 111–114
  fakes, 124–126
  mock objects, 108–110, 121–123
  spies, 118–120
  structure of, 109–110
  stubs, 115–117
Test-driven development (TDD)
  algorithm simplicity via, 61–62, 78–79
  Arrange/Act/Assert pattern in, 104
  BDD as, 105
  benefits of, 26–34
  continuous integration and, 337
  as design technique, 312
  documentation in, 26–28
  as double-entry bookkeeping, 23
  examples, 36–79
  as foundational discipline, 14, 15–16
  four laws of, 23–35
  fragile test problem and, 160–161
  getting stuck with, 95–103
  incremental complexity in, 92
  long-term mastery of, 148
  necessity for, 305–306
  proofs of, 323–325
  test suite holes, 28–29
  two schools of, 140–143
  uncertainty principle, 126–139
Test-Specific Subclass pattern, 155–156
Tests
  acceptance, 18, 246–249
  Arrange/Act/Assert pattern, 103–104
  assertions and spy behavior, 123
  automated, 246, 267–269, 340
  code and test communication, 236
  continuous running of, 249
  coverage by, 229–231, 239, 343–344
  for databases, 148–150
  decoupling production code from, 161–184, 218
  degenerate, 53–54, 57, 66
  deleting stairstep, 66
  design of, 160–184
  excluding production data from, 116
  as finite state transitions, 107
  fragile, 59, 139–140, 160–161
  functions constrained by, 127
  for GUIs, 150–154
  historic, 226
  integration, 182–183
  mutation testing, 344
  patterns, 100, 155–160
  to prevent harm, 305–306
  programmer, 182–183
  as refactoring discipline, 218, 219–220, 221
  rule of simplicity for, 95–96
  semantic stability of, 344–345
specificity of, for generic code, 59, 82, 183–184
speeding up, 348–349
streamlining, 113
as theories, 323
transformations for passing, 184–196
vs. placing hooks, 227
written by/for BA and QA, 248–249
Thompson, Ken, 284
ThoughtWorks, 142
Tichy, Walter, 333
Time management, 352–354
Toggles, branches vs., 338–340
Toyota, 300–301, 302
Transformation priority premise, 195–196
Transformations, 184–196
Transistors, 6
Transition triplets, 106
Traversal code, 237–238
Trim call, 101
Turing, Alan, 5–6, 62, 280–281

U
UML diagrams, 64, 73, 144
Uncertainty principle, 126–139, 140
Unconditional to selection transformation, 188–189
Underflow exceptions, 47
UNIX, 284
Untangling software, 224
Unwound loops, 57, 60
User interface
and the Chicago school, 142
and the London school, 141
product as changed via, 304, 308
and test automation, 268–269
testing the, 269–270, 348
User perspective, 252

V
Vacuum tube computer, 6
Values
  testing, 140
  value to list transformation, 189
  value to mutated value transformation, 190–191
Variables, 202–204, 235
View model, 151
Virtual teamwork, 356–358
Viscardi, Stacia, xix
Viscosity, 347–349
Volkswagen, 289, 296, 297
Von Neumann, Jon, 282

W
Wake, Bill, 103
While statements, 58, 60, 202
Whitehead, Gustave, 3
Work, best, 306–315
Work breakdown structure (WBS), 366–367
World War II, 3, 5
Wrap call, 100
Wright brothers, 3

X
XRay class, 155
XUnit Test Patterns (Meszaros), 110, 154

Y
YAGNI, 226–228

Z
Zuse, Konrad, 5