

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship



Foreword by James O. Coplien

Robert C. Martin Series

PRENTICE
HALL

2
eBooks

The Clean Coder

A Code of Conduct for Professional Programmers



Foreword by Matthew Heusser, Software Process Naturalist

Robert C. Martin

**THE ROBERT C. MARTIN
CLEAN CODE COLLECTION**

The Robert C. Martin Clean Code Collection



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

Note from the Publisher

The *Robert C. Martin Clean Code Collection* consists of two bestselling eBooks:

- *Clean Code: A Handbook of Agile Software Craftsmanship*
- *The Clean Coder: A Code of Conduct for Professional Programmers*

In this collection, Robert C. Martin, also known as “Uncle Bob,” provides a pragmatic method for writing better code from the start. He reveals the disciplines, techniques, tools, and practices that separate software craftsmen from mere “9-to-5” programmers. Within this collection are the tools and methods you need to become a true software professional.

To simplify access to each book, we’ve appended “A” to the pages of *Clean Code: A Handbook of Agile Software Craftsmanship*, and “B” to pages of *The Clean Coder: A Code of Conduct for Professional Programmers*. This enabled us to produce a single, comprehensive table of contents and dedicated indexes.

We hope you find this collection useful!

—The editorial and production teams at Prentice Hall

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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: informit.com/ph

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-292847-2

ISBN-10: 0-13-292847-7

Table of Contents

CLEAN CODE

1 Clean Code1A
There Will Be Code2A
Bad Code3A
The Total Cost of Owning a Mess4A
The Grand Redesign in the Sky5A
Attitude5A
The Primal Conundrum6A
The Art of Clean Code?6A
What Is Clean Code?7A
Schools of Thought12A
We Are Authors13A
The Boy Scout Rule14A
Prequel and Principles15A
Conclusion15A
Bibliography15A
2 Meaningful Names17A
Introduction17A
Use Intention-Revealing Names18A
Avoid Disinformation19A
Make Meaningful Distinctions20A
Use Pronounceable Names21A
Use Searchable Names22A
Avoid Encodings23A
Hungarian Notation23A
Member Prefixes24A
Interfaces and Implementations24A
Avoid Mental Mapping25A
Class Names25A
Method Names25A
Don't Be Cute26A
Pick One Word per Concept26A
Don't Pun26A

Use Solution Domain Names	27A
Use Problem Domain Names	27A
Add Meaningful Context	27A
Don't Add Gratuitous Context	29A
Final Words	30A
3 Functions	31A
Small!	34A
Blocks and Indenting	35A
Do One Thing	35A
Sections within Functions	36A
One Level of Abstraction per Function	36A
Reading Code from Top to Bottom: The Stepdown Rule	37A
Switch Statements	37A
Use Descriptive Names	39A
Function Arguments	40A
Common Monadic Forms	41A
Flag Arguments	41A
Dyadic Functions	42A
Triads	42A
Argument Objects	43A
Argument Lists	43A
Verbs and Keywords	43A
Have No Side Effects	44A
Output Arguments	45A
Command Query Separation	45A
Prefer Exceptions to Returning Error Codes	46A
Extract Try/Catch Blocks	46A
Error Handling Is One Thing	47A
The <code>Error.java</code> Dependency Magnet	47A
Don't Repeat Yourself	48A
Structured Programming	48A
How Do You Write Functions Like This?	49A
Conclusion	49A
SetupTeardownIncluder	50A
Bibliography	52A

4 Comments	53A
Comments Do Not Make Up for Bad Code	55A
Explain Yourself in Code	55A
Good Comments	55A
Legal Comments	55A
Informative Comments	56A
Explanation of Intent	56A
Clarification	57A
Warning of Consequences	58A
TODO Comments	58A
Amplification	59A
Javadocs in Public APIs	59A
Bad Comments	59A
Mumbling	59A
Redundant Comments	60A
Misleading Comments	63A
Mandated Comments	63A
Journal Comments	63A
Noise Comments	64A
Scary Noise	66A
Don't Use a Comment When You Can Use a Function or a Variable	67A
Position Markers	67A
Closing Brace Comments	67A
Attributions and Bylines	68A
Commented-Out Code	68A
HTML Comments	69A
Nonlocal Information	69A
Too Much Information	70A
Inobvious Connection	70A
Function Headers	70A
Javadocs in Nonpublic Code	71A
Example	71A
Bibliography	74A
 5 Formatting	 75A
The Purpose of Formatting	76A
Vertical Formatting	76A

The Newspaper Metaphor	77A
Vertical Openness Between Concepts	78A
Vertical Density	79A
Vertical Distance	80A
Vertical Ordering	84A
Horizontal Formatting	85A
Horizontal Openness and Density	86A
Horizontal Alignment	87A
Indentation	88A
Dummy Scopes	90A
Team Rules	90A
Uncle Bob's Formatting Rules	90A
6 Objects and Data Structures	93A
Data Abstraction	93A
Data/Object Anti-Symmetry	95A
The Law of Demeter	97A
Train Wrecks	98A
Hybrids	99A
Hiding Structure	99A
Data Transfer Objects	100A
Active Record	101A
Conclusion	101A
Bibliography	101A
7 Error Handling	103A
Use Exceptions Rather Than Return Codes	104A
Write Your Try-Catch-Finally Statement First	105A
Use Unchecked Exceptions	106A
Provide Context with Exceptions	107A
Define Exception Classes in Terms of a Caller's Needs	107A
Define the Normal Flow	109A
Don't Return Null	110A
Don't Pass Null	111A
Conclusion	112A
Bibliography	112A

8 Boundaries	113A
Using Third-Party Code	.114A
Exploring and Learning Boundaries	.116A
Learning log4j	.116A
Learning Tests Are Better Than Free	.118A
Using Code That Does Not Yet Exist	.118A
Clean Boundaries	.120A
Bibliography	.120A
9 Unit Tests	121A
The Three Laws of TDD	.122A
Keeping Tests Clean	.123A
Tests Enable the -ilities	.124A
Clean Tests	.124A
Domain-Specific Testing Language	.127A
A Dual Standard	.127A
One Assert per Test	.130A
Single Concept per Test	.131A
F.I.R.S.T.	.132A
Conclusion	.133A
Bibliography	.133A
10 Classes	135A
Class Organization	.136A
Encapsulation	.136A
Classes Should Be Small!	.136A
The Single Responsibility Principle	.138A
Cohesion	.140A
Maintaining Cohesion Results in Many Small Classes	.141A
Organizing for Change	.147A
Isolating from Change	.149A
Bibliography	.151A
11 Systems	153A
How Would You Build a City?	.154A
Separate Constructing a System from Using It	.154A
Separation of Main	.155A

Factories	155A
Dependency Injection	157A
Scaling Up	157A
Cross-Cutting Concerns	160A
Java Proxies	161A
Pure Java AOP Frameworks	163A
AspectJ Aspects	166A
Test Drive the System Architecture	166A
Optimize Decision Making	167A
Use Standards Wisely, When They Add Demonstrable Value	168A
Systems Need Domain-Specific Languages	168A
Conclusion	169A
Bibliography	169A
12 Emergence	171A
Getting Clean via Emergent Design	171A
Simple Design Rule 1: Runs All the Tests	172A
Simple Design Rules 2–4: Refactoring	172A
No Duplication	173A
Expressive	175A
Minimal Classes and Methods	176A
Conclusion	176A
Bibliography	176A
13 Concurrency	177A
Why Concurrency?	178A
Myths and Misconceptions	179A
Challenges	180A
Concurrency Defense Principles	180A
Single Responsibility Principle	181A
Corollary: Limit the Scope of Data	181A
Corollary: Use Copies of Data	181A
Corollary: Threads Should Be as Independent as Possible	182A
Know Your Library	182A
Thread-Safe Collections	182A

Know Your Execution Models	183A
Producer-Consumer	184A
Readers-Writers	184A
Dining Philosophers	184A
Beware Dependencies Between Synchronized Methods	185A
Keep Synchronized Sections Small	185A
Writing Correct Shut-Down Code Is Hard	186A
Testing Threaded Code	186A
Treat Spurious Failures as Candidate Threading Issues	187A
Get Your Nonthreaded Code Working First	187A
Make Your Threaded Code Pluggable	187A
Make Your Threaded Code Tunable	187A
Run with More Threads Than Processors	188A
Run on Different Platforms	188A
Instrument Your Code to Try and Force Failures	188A
Hand-Coded	189A
Automated	189A
Conclusion	190A
Bibliography	191A
14 Successive Refinement	193A
Args Implementation	194A
How Did I Do This?	200A
Args: The Rough Draft	201A
So I Stopped	212A
On Incrementalism	212A
String Arguments	214A
Conclusion	250A
15 JUnit Internals	251A
The JUnit Framework	252A
Conclusion	265A
16 Refactoring <code>Serializable</code>	267A
First, Make It Work	268A
Then Make It Right	270A

Conclusion	284A
Bibliography	284A
17 Smells and Heuristics	285A
Comments	286A
C1: Inappropriate Information	286A
C2: Obsolete Comment	286A
C3: Redundant Comment	286A
C4: Poorly Written Comment	287A
C5: Commented-Out Code	287A
Environment	287A
E1: Build Requires More Than One Step	287A
E2: Tests Require More Than One Step	287A
Functions	288A
F1: Too Many Arguments	288A
F2: Output Arguments	288A
F3: Flag Arguments	288A
F4: Dead Function	288A
General	288A
G1: Multiple Languages in One Source File	288A
G2: Obvious Behavior Is Unimplemented	288A
G3: Incorrect Behavior at the Boundaries	289A
G4: Overridden Safeties	289A
G5: Duplication	289A
G6: Code at Wrong Level of Abstraction	290A
G7: Base Classes Depending on Their Derivatives	291A
G8: Too Much Information	291A
G9: Dead Code	292A
G10: Vertical Separation	292A
G11: Inconsistency	292A
G12: Clutter	293A
G13: Artificial Coupling	293A
G14: Feature Envy	293A
G15: Selector Arguments	294A
G16: Obscured Intent	295A
G17: Misplaced Responsibility	295A
G18: Inappropriate Static	296A

G19: Use Explanatory Variables	296A
G20: Function Names Should Say What They Do	297A
G21: Understand the Algorithm	297A
G22: Make Logical Dependencies Physical	298A
G23: Prefer Polymorphism to If/Else or Switch/Case	299A
G24: Follow Standard Conventions	299A
G25: Replace Magic Numbers with Named Constants	300A
G26: Be Precise	301A
G27: Structure over Convention	301A
G28: Encapsulate Conditionals	301A
G29: Avoid Negative Conditionals	302A
G30: Functions Should Do One Thing	302A
G31: Hidden Temporal Couplings	302A
G32: Don't Be Arbitrary	303A
G33: Encapsulate Boundary Conditions	304A
G34: Functions Should Descend Only One Level of Abstraction	304A
G35: Keep Configurable Data at High Levels	306A
G36: Avoid Transitive Navigation	306A
Java	307A
J1: Avoid Long Import Lists by Using Wildcards	307A
J2: Don't Inherit Constants	307A
J3: Constants versus Enums	308A
Names	309A
N1: Choose Descriptive Names	309A
N2: Choose Names at the Appropriate Level of Abstraction	311A
N3: Use Standard Nomenclature Where Possible	311A
N4: Unambiguous Names	312A
N5: Use Long Names for Long Scopes	312A
N6: Avoid Encodings	312A
N7: Names Should Describe Side-Effects.	313A

Tests	313A
T1: Insufficient Tests	313A
T2: Use a Coverage Tool!	313A
T3: Don't Skip Trivial Tests	313A
T4: An Ignored Test Is a Question about an Ambiguity	313A
T5: Test Boundary Conditions	314A
T6: Exhaustively Test Near Bugs	314A
T7: Patterns of Failure Are Revealing	314A
T8: Test Coverage Patterns Can Be Revealing	314A
T9: Tests Should Be Fast	314A
Conclusion	314A
Bibliography	315A
A Concurrency II	317A
Client/Server Example	317A
The Server	317A
Adding Threading	319A
Server Observations	319A
Conclusion	321A
Possible Paths of Execution	321A
Number of Paths	322A
Digging Deeper	323A
Conclusion	326A
Knowing Your Library	326A
Executor Framework	326A
Nonblocking Solutions	327A
Nonthread-Safe Classes	328A
Dependencies Between Methods Can Break Concurrent Code	329A
Tolerate the Failure	330A
Client-Based Locking	330A
Server-Based Locking	332A
Increasing Throughput	333A
Single-Thread Calculation of Throughput	334A
Multithread Calculation of Throughput	335A

Deadlock335A
Mutual Exclusion336A
Lock & Wait337A
No Preemption337A
Circular Wait337A
Breaking Mutual Exclusion337A
Breaking Lock & Wait338A
Breaking Preemption338A
Breaking Circular Wait338A
Testing Multithreaded Code339A
Tool Support for Testing Thread-Based Code342A
Conclusion	342A
Tutorial: Full Code Examples343A
Client/Server Nonthreaded343A
Client/Server Using Threads347A
B org.jfree.date.SerialDate	349A
C Cross References of Heuristics	409A
Epilogue	411A
Index	413A

THE CLEAN CODER

Pre-Requisite Introduction	1B
1 Professionalism	7B
Be Careful What You Ask For8B
Taking Responsibility8B
First, Do No Harm11B
Work Ethic16B
Bibliography22B
2 Saying No	23B
Adversarial Roles26B
High Stakes29B

Being a “Team Player”	30B
The Cost of Saying Yes	36B
Code Impossible	41B
3 Saying Yes	45B
A Language of Commitment	47B
Learning How to Say “Yes”	52B
Conclusion	56B
4 Coding	57B
Preparedness	58B
The Flow Zone	62B
Writer’s Block	64B
Debugging	66B
Pacing Yourself	69B
Being Late	71B
Help	73B
Bibliography	76B
5 Test Driven Development	77B
The Jury Is In	79B
The Three Laws of TDD	79B
What TDD Is Not	83B
Bibliography	84B
6 Practicing	85B
Some Background on Practicing	86B
The Coding Dojo	89B
Broadening Your Experience	93B
Conclusion	94B
Bibliography	94B
7 Acceptance Testing	95B
Communicating Requirements	95B
Acceptance Tests	100B
Conclusion	111B

8 Testing Strategies	113B
QA Should Find Nothing	114B
The Test Automation Pyramid	115B
Conclusion	119B
Bibliography	119B
9 Time Management	121B
Meetings	122B
Focus-Manna	127B
Time Boxing and Tomatoes	130B
Avoidance	131B
Blind Alleys	131B
Marshes, Bogs, Swamps, and Other Messes	132B
Conclusion	133B
10 Estimation	135B
What Is an Estimate?	138B
PERT	141B
Estimating Tasks	144B
The Law of Large Numbers	147B
Conclusion	147B
Bibliography	148B
11 Pressure	149B
Avoiding Pressure	151B
Handling Pressure	153B
Conclusion	155B
12 Collaboration	157B
Programmers versus People	159B
Cerebellums	164B
Conclusion	166B
13 Teams and Projects	167B
Does It Blend?	168B
Conclusion	171B
Bibliography	171B

14 Mentoring, Apprenticeship, and Craftsmanship	173B
Degrees of Failure	174B
Mentoring	174B
Apprenticeship	180B
Craftsmanship	184B
Conclusion	185B
A Tooling	187B
Tools	189B
Source Code Control	189B
IDE/Editor	194B
Issue Tracking	196B
Continuous Build	197B
Unit Testing Tools	198B
Component Testing Tools	199B
Integration Testing Tools	200B
UML/MDA	201B
Conclusion	204B
Index	205B

This page intentionally left blank

Clean Code

A Handbook of Agile Software Craftsmanship

The Object Mentors:

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger
Jeffrey J. Langr Brett L. Schuchert
James W. Grenning Kevin Dean Wampler
Object Mentor Inc.

*Writing clean code is what you must do in order to call yourself a professional.
There is no reasonable excuse for doing anything less than your best.*


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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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@pearsoned.com

Visit us on the Web: informati.com/ph

Library of Congress Cataloging-in-Publication Data

Martin, Robert C.

Clean code : a handbook of agile software craftsmanship / Robert C. Martin.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-235088-2 (pbk. : alk. paper)

1. Agile software development. 2. Computer software—Reliability. I. Title.

QA76.76.D47M3652 2008

005.1—dc22

2008024750

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-13-235088-4

ISBN-10: 0-13-235088-2

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

Ninth printing April, 2011

For Ann Marie: The ever enduring love of my life.

Foreword

One of our favorite candies here in Denmark is Ga-Jol, whose strong licorice vapors are a perfect complement to our damp and often chilly weather. Part of the charm of Ga-Jol to us Danes is the wise or witty sayings printed on the flap of every box top. I bought a two-pack of the delicacy this morning and found that it bore this old Danish saw:

Ærlighed i små ting er ikke nogen lille ting.

“Honesty in small things is not a small thing.” It was a good omen consistent with what I already wanted to say here. Small things matter. This is a book about humble concerns whose value is nonetheless far from small.

God is in the details, said the architect Ludwig mies van der Rohe. This quote recalls contemporary arguments about the role of architecture in software development, and particularly in the Agile world. Bob and I occasionally find ourselves passionately engaged in this dialogue. And yes, mies van der Rohe was attentive to utility and to the timeless forms of building that underlie great architecture. On the other hand, he also personally selected every doorknob for every house he designed. Why? Because small things matter.

In our ongoing “debate” on TDD, Bob and I have discovered that we agree that software architecture has an important place in development, though we likely have different visions of exactly what that means. Such quibbles are relatively unimportant, however, because we can accept for granted that responsible professionals give some time to thinking and planning at the outset of a project. The late-1990s notions of design driven only by the tests and the code are long gone. Yet attentiveness to detail is an even more critical foundation of professionalism than is any grand vision. First, it is through practice in the small that professionals gain proficiency and trust for practice in the large. Second, the smallest bit of sloppy construction, of the door that does not close tightly or the slightly crooked tile on the floor, or even the messy desk, completely dispels the charm of the larger whole. That is what clean code is about.

Still, architecture is just one metaphor for software development, and in particular for that part of software that delivers the initial product in the same sense that an architect delivers a pristine building. In these days of Scrum and Agile, the focus is on quickly bringing product to market. We want the factory running at top speed to produce software. These are human factories: thinking, feeling coders who are working from a product backlog or user story to create product. The manufacturing metaphor looms ever strong in such thinking. The production aspects of Japanese auto manufacturing, of an assembly-line world, inspire much of Scrum.

Yet even in the auto industry, the bulk of the work lies not in manufacturing but in maintenance—or its avoidance. In software, 80% or more of what we do is quaintly called “maintenance”: the act of repair. Rather than embracing the typical Western focus on *producing* good software, we should be thinking more like home repairmen in the building industry, or auto mechanics in the automotive field. What does Japanese management have to say about *that*?

In about 1951, a quality approach called Total Productive Maintenance (TPM) came on the Japanese scene. Its focus is on maintenance rather than on production. One of the major pillars of TPM is the set of so-called 5S principles. 5S is a set of disciplines—and here I use the term “discipline” instructively. These 5S principles are in fact at the foundations of Lean—another buzzword on the Western scene, and an increasingly prominent buzzword in software circles. These principles are not an option. As Uncle Bob relates in his front matter, good software practice requires such discipline: focus, presence of mind, and thinking. It is not always just about doing, about pushing the factory equipment to produce at the optimal velocity. The 5S philosophy comprises these concepts:

- *Seiri*, or organization (think “sort” in English). Knowing where things are—using approaches such as **suitable** naming—is **crucial**. You think naming identifiers isn’t important? Read on in the following chapters.
- *Seiton*, or tidiness (think “systematize” in English). There is an old American saying: *A place for everything, and everything in its place*. A piece of code should be where you expect to find it—and, if not, you should re-factor to get it there.
- *Seiso*, or cleaning (think “shine” in English): Keep the workplace free of hanging wires, grease, scraps, and waste. What do the authors here say about littering your code with comments and commented-out code lines that capture history or wishes for the future? Get rid of them.
- *Seiketsu*, or standardization: The group agrees about how to keep the workplace clean. Do you think this book says anything about having a consistent coding style and set of practices within the group? Where do those standards come from? Read on.
- *Shitsuke*, or discipline (*self*-discipline). This means having the discipline to follow the practices and to frequently reflect on one’s work and be willing to change.

If you **take up** the challenge—yes, the challenge—of reading and applying this book, you’ll come to understand and appreciate the last point. Here, we are finally driving to the roots of responsible professionalism in a profession that should be concerned with the life cycle of a product. As we maintain automobiles and other machines under TPM, breakdown maintenance—waiting for bugs to surface—is the exception. Instead, we go up a level: inspect the machines every day and fix wearing parts before they break, or do the equivalent of the proverbial 10,000-mile oil change to forestall wear and tear. In code, refactor mercilessly. You can improve yet one level further, as the TPM movement innovated over 50 years ago: build machines that are more maintainable in the first place. Making your code readable is as important as making it executable. The ultimate practice, introduced in TPM circles around 1960, is to focus on introducing entire new machines or

replacing old ones. As Fred Brooks admonishes us, we should probably re-do major software chunks from scratch every seven years or so to sweep away creeping cruft. Perhaps we should update Brooks' time constant to an order of weeks, days or hours instead of years. That's where detail lies.

There is great power in detail, yet there is something humble and profound about this approach to life, as we might stereotypically expect from any approach that claims Japanese roots. But this is not only an Eastern outlook on life; English and American folk wisdom are full of such admonishments. The Seiton quote from above flowed from the pen of an Ohio minister who literally viewed neatness “as a remedy for every degree of evil.” How about Seiso? *Cleanliness is next to godliness*. As beautiful as a house is, a messy desk robs it of its splendor. How about Shutsuke in these small matters? *He who is faithful in little is faithful in much*. How about being eager to re-factor at the responsible time, strengthening one’s position for subsequent “big” decisions, rather than putting it off? *A stitch in time saves nine. The early bird catches the worm. Don’t put off until tomorrow what you can do today*. (Such was the original sense of the phrase “the last responsible moment” in Lean until it fell into the hands of software consultants.) How about calibrating the place of small, individual efforts in a grand whole? *Mighty oaks from little acorns grow*. Or how about integrating simple preventive work into everyday life? *An ounce of prevention is worth a pound of cure. An apple a day keeps the doctor away*. Clean code honors the deep roots of wisdom beneath our broader culture, or our culture as it once was, or should be, and *can* be with attentiveness to detail.

Even in the grand architectural literature we find saws that hark back to these supposed details. Think of mies van der Rohe’s doorknobs. That’s *seiri*. That’s being attentive to every variable name. You should name a variable using the same care with which you name a first-born child.

As every homeowner knows, such care and ongoing refinement never come to an end. The architect Christopher Alexander—father of patterns and pattern languages—views every act of design itself as a small, local act of repair. And he views the craftsmanship of fine structure to be the sole purview of the architect; the larger forms can be left to patterns and their application by the inhabitants. Design is ever ongoing not only as we add a new room to a house, but as we are attentive to repainting, replacing worn carpets, or upgrading the kitchen sink. Most arts echo analogous sentiments. In our search for others who ascribe God’s home as being in the details, we find ourselves in the good company of the 19th century French author Gustav Flaubert. The French poet Paul Valery advises us that a poem is never done and bears continual rework, and to stop working on it is abandonment. Such preoccupation with detail is common to all endeavors of excellence. So maybe there is little new here, but in reading this book you will be challenged to take up good disciplines that you long ago surrendered to apathy or a desire for spontaneity and just “responding to change.”

Unfortunately, we usually don’t view such concerns as key cornerstones of the art of programming. We abandon our code early, not because it is done, but because our value system focuses more on outward appearance than on the substance of what we deliver.

This inattentiveness costs us in the end: *A bad penny always shows up*. Research, neither in industry nor in academia, humbles itself to the lowly station of keeping code clean. Back in my days working in the Bell Labs Software Production Research organization (*Production*, indeed!) we had some back-of-the-envelope findings that suggested that consistent indentation style was one of the most statistically significant indicators of low bug density. We want it to be that architecture or programming language or some other high notion should be the cause of quality; as people whose supposed professionalism owes to the mastery of tools and lofty design methods, we feel insulted by the value that those factory-floor machines, the coders, add through the simple consistent application of an indentation style. To quote my own book of 17 years ago, such style distinguishes excellence from mere competence. The Japanese worldview understands the crucial value of the everyday worker and, more so, of the systems of development that owe to the simple, everyday actions of those workers. Quality is the result of a million selfless acts of care—not just of any great method that descends from the heavens. That these acts are simple doesn't mean that they are simplistic, and it hardly means that they are easy. They are nonetheless the fabric of greatness and, more so, of beauty, in any human endeavor. To ignore them is not yet to be fully human.

Of course, I am still an advocate of thinking at broader scope, and particularly of the value of architectural approaches rooted in deep domain knowledge and software usability. The book isn't about that—or, at least, it isn't obviously about that. This book has a subtler message whose profoundness should not be underappreciated. It fits with the current saw of the really code-based people like Peter Sommerlad, Kevlin Henney and Giovanni Asproni. “The code is the design” and “Simple code” are their mantras. While we must take care to remember that the interface is the program, and that its structures have much to say about our program structure, it is crucial to continuously adopt the humble stance that the design lives in the code. And while rework in the manufacturing metaphor leads to cost, rework in design leads to value. We should view our code as the beautiful articulation of noble efforts of design—design as a process, not a static endpoint. It's in the code that the architectural metrics of coupling and cohesion play out. If you listen to Larry Constantine describe coupling and cohesion, he speaks in terms of code—not lofty abstract concepts that one might find in UML. Richard Gabriel advises us in his essay, “Abstraction Descant” that abstraction is evil. Code is anti-evil, and clean code is perhaps divine.

Going back to my little box of Ga-Jol, I think it's important to note that the Danish wisdom advises us not just to pay attention to small things, but also to be *honest* in small things. This means being honest to the code, honest to our colleagues about the state of our code and, most of all, being honest with ourselves about our code. Did we Do our Best to “leave the campground cleaner than we found it”? Did we re-factor our code before checking in? These are not peripheral concerns but concerns that lie squarely in the center of Agile values. It is a recommended practice in Scrum that re-factoring be part of the concept of “Done.” Neither architecture nor clean code insist on perfection, only on honesty and doing the best we can. *To err is human; to forgive, divine*. In Scrum, we make everything visible. We air our dirty laundry. We are honest about the state of our code because

code is never perfect. We become more fully human, more worthy of the divine, and closer to that greatness in the details.

In our profession, we desperately need all the help we can get. If a clean shop floor reduces accidents, and well-organized shop tools increase productivity, then I'm all for them. As for this book, it is the best pragmatic application of Lean principles to software I have ever seen in print. I expected no less from this practical little group of thinking individuals that has been striving together for years not only to become better, but also to gift their knowledge to the industry in works such as you now find in your hands. It leaves the world a little better than I found it before Uncle Bob sent me the manuscript.

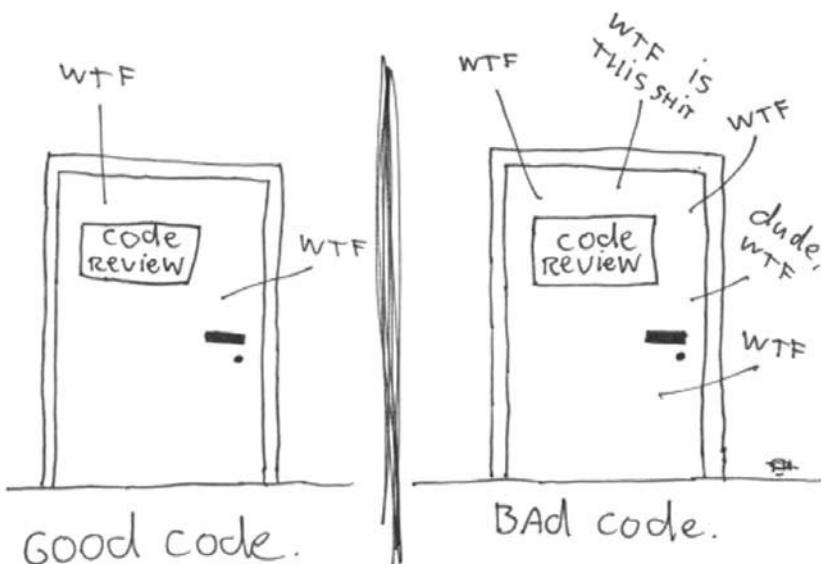
Having completed this exercise in lofty insights, I am off to clean my desk.

James O. Coplien
Mørdrup, Denmark

This page intentionally left blank

Introduction

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/minute



(c) 2008 Focus Shift

Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just a normal code review or have we found a stream of horrible problems shortly after going live? Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and managers breathing down

our necks? How can we make sure we wind up behind the *right* door when the going gets tough? The answer is: *craftsmanship*.

There are two parts to learning craftsmanship: knowledge and work. You must gain the knowledge of principles, patterns, practices, and heuristics that a craftsman knows, and you must also grind that knowledge into your fingers, eyes, and gut by working hard and practicing.

I can teach you the physics of riding a bicycle. Indeed, the classical mathematics is relatively straightforward. Gravity, friction, angular momentum, center of mass, and so forth, can be demonstrated with less than a page full of equations. Given those formulae I could prove to you that bicycle riding is practical and give you all the knowledge you needed to make it work. And you'd still fall down the first time you climbed on that bike.

Coding is no different. We could write down all the “feel good” principles of clean code and then trust you to do the work (in other words, let you fall down when you get on the bike), but then what kind of teachers would that make us, and what kind of student would that make you?

No. That's not the way this book is going to work.

Learning to write clean code is *hard work*. It requires more than just the knowledge of principles and patterns. You must *sweat* over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.

Be prepared to work hard while reading this book. This is not a “feel good” book that you can read on an airplane and finish before you land. This book will make you work, *and work hard*. What kind of work will you be doing? You'll be reading code—lots of code. And you will be challenged to think about what's right about that code and what's wrong with it. You'll be asked to follow along as we take modules apart and put them back together again. This will take time and effort; but we think it will be worth it.

We have divided this book into three parts. The first several chapters describe the principles, patterns, and practices of writing clean code. There is quite a bit of code in these chapters, and they will be challenging to read. They'll prepare you for the second section to come. If you put the book down after reading the first section, good luck to you!

The second part of the book is the harder work. It consists of several case studies of ever-increasing complexity. Each case study is an exercise in cleaning up some code—of transforming code that has some problems into code that has fewer problems. The detail in this section is *intense*. You will have to flip back and forth between the narrative and the code listings. You will have to analyze and understand the code we are working with and walk through our reasoning for making each change we make. Set aside some time because *this should take you days*.

The third part of this book is the payoff. It is a single chapter containing a list of heuristics and smells gathered while creating the case studies. As we walked through and cleaned up the code in the case studies, we documented every reason for our actions as a

heuristic or smell. We tried to understand our own reactions to the code we were reading and changing, and worked hard to capture why we felt what we felt and did what we did. The result is a knowledge base that describes the way we think when we write, read, and clean code.

This knowledge base is of limited value if you don't do the work of carefully reading through the case studies in the second part of this book. In those case studies we have carefully annotated each change we made with forward references to the heuristics. These forward references appear in square brackets like this: [H22]. This lets you see the *context* in which those heuristics were applied and written! It is not the heuristics themselves that are so valuable, it is the *relationship between those heuristics and the discrete decisions we made while cleaning up the code in the case studies*.

To further help you with those relationships, we have placed a cross-reference at the end of the book that shows the page number for every forward reference. You can use it to look up each place where a certain heuristic was applied.

If you read the first and third sections and skip over the case studies, then you will have read yet another “feel good” book about writing good software. But if you take the time to work through the case studies, following every tiny step, every minute decision—if you put yourself in our place, and force yourself to think along the same paths that we thought, then you will gain a much richer understanding of those principles, patterns, practices, and heuristics. They won’t be “feel good” knowledge any more. They’ll have been ground into your gut, fingers, and heart. They’ll have become part of you in the same way that a bicycle becomes an extension of your will when you have mastered how to ride it.

Acknowledgments

Thank you to my two artists, Jeniffer Kohnke and Angela Brooks. Jennifer is responsible for the stunning and creative pictures at the start of each chapter and also for the portraits of Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers, and myself.

Angela is responsible for the clever pictures that adorn the innards of each chapter. She has done quite a few pictures for me over the years, including many of the inside pictures in *Agile Software Development: Principles, Patterns, and Practices*. She is also my firstborn in whom I am well pleased.

A special thanks goes out to my reviewers Bob Bogetti, George Bullock, Jeffrey Overbey, and especially Matt Heusser. They were brutal. They were cruel. They were relentless. They pushed me hard to make necessary improvements.

Thanks to my publisher, Chris Guzikowski, for his support, encouragement, and jovial countenance. Thanks also to the editorial staff at Pearson, including Raina Chrobak for keeping me honest and punctual.

Thanks to Micah Martin, and all the guys at 8th Light (www.8thlight.com) for their reviews and encouragement.

Thanks to all the Object Mentors, past, present, and future, including: Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottinger, Joleen Craig, Janice Brown, Susan Rosso, et al.

Thanks to Jim Newkirk, my friend and business partner, who taught me more than I think he realizes. Thanks to Kent Beck, Martin Fowler, Ward Cunningham, Bjarne Stroustrup, Grady Booch, and all my other mentors, compatriots, and foils. Thanks to John Vlissides for being there when it counted. Thanks to the guys at Zebra for allowing me to rant on about how long a function should be.

And, finally, thank you for reading these thank yous.

On the Cover

The image on the cover is M104: The Sombrero Galaxy. M104 is located in Virgo and is just under 30 million light-years from us. At its core is a supermassive black hole weighing in at about a billion solar masses.

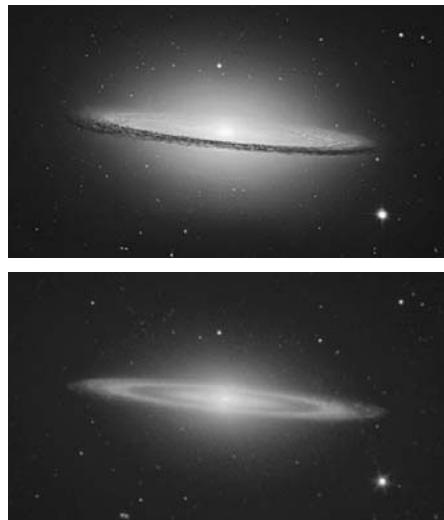
Does the image remind you of the explosion of the Klingon power moon *Praxis*? I vividly remember the scene in *Star Trek VI* that showed an equatorial ring of debris flying away from that explosion. Since that scene, the equatorial ring has been a common artifact in sci-fi movie explosions. It was even added to the explosion of Alderaan in later editions of the first *Star Wars* movie.

What caused this ring to form around M104? Why does it have such a huge central bulge and such a bright and tiny nucleus? It looks to me as though the central black hole lost its cool and blew a 30,000 light-year hole in the middle of the galaxy. Woe befell any civilizations that might have been in the path of that cosmic disruption.

Supermassive black holes swallow whole stars for lunch, converting a sizeable fraction of their mass to energy. $E = MC^2$ is leverage enough, but when M is a stellar mass: Look out! How many stars fell headlong into that maw before the monster was satiated? Could the size of the central void be a hint?

The image of M104 on the cover is a combination of the famous visible light photograph from Hubble (right), and the recent infrared image from the Spitzer orbiting observatory (below, right). It's the infrared image that clearly shows us the ring nature of the galaxy. In visible light we only see the front edge of the ring in silhouette. The central bulge obscures the rest of the ring.

But in the infrared, the hot particles in the ring shine through the central bulge. The two images combined give us a view we've not seen before and imply that long ago it was a raging inferno of activity.



Cover image: © Spitzer Space Telescope

Clean Code



You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.

This is a book about good programming. It is filled with code. We are going to look at code from every different direction. We'll look down at it from the top, up at it from the bottom, and through it from the inside out. By the time we are done, we're going to know a lot about code. What's more, we'll be able to tell the difference between good code and bad code. We'll know how to write good code. And we'll know how to transform bad code into good code.

There Will Be Code

One might argue that a book about code is somehow behind the times—that code is no longer the issue; that we should be concerned about models and requirements instead. Indeed some have suggested that we are close to the end of code. That soon all code will be generated instead of written. That programmers simply won't be needed because business people will generate programs from specifications.

Nonsense! We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such detail that a machine can execute them *is programming*. Such a specification *is code*.

I expect that the level of abstraction of our languages will continue to increase. I also expect that the number of domain-specific languages will continue to grow. This will be a good thing. But it will not eliminate code. Indeed, all the specifications written in these higher level and domain-specific language will *be code!* It will still need to be rigorous, accurate, and so formal and detailed that a machine can understand and execute it.

The folks who think that code will one day disappear are like mathematicians who hope one day to discover a mathematics that does not have to be formal. They are hoping that one day we will discover a way to create machines that can do what we want rather than what we say. These machines will have to be able to understand us so well that they can translate vaguely specified needs into perfectly executing programs that precisely meet those needs.

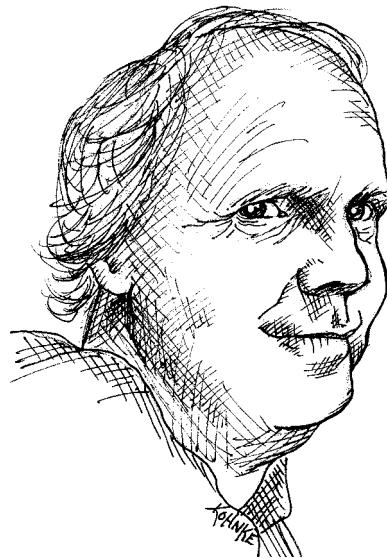
This will never happen. Not even humans, with all their intuition and creativity, have been able to create successful systems from the vague feelings of their customers. Indeed, if the discipline of requirements specification has taught us anything, it is that well-specified requirements are as formal as code and can act as executable tests of that code!

Remember that code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision—so there will always be code.

Bad Code

I was recently reading the preface to Kent Beck's book *Implementation Patterns*.¹ He says, ". . . this book is based on a rather fragile premise: that good code matters. . . ." A *fragile* premise? I disagree! I think that premise is one of the most robust, supported, and overloaded of all the premises in our craft (and I think Kent knows it). We know good code matters because we've had to deal for so long with its lack.

I know of one company that, in the late 80s, wrote a *killer* app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.



Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. *It was the bad code that brought the company down.*

Have you ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. Indeed, we have a name for it. We call it *wading*. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle to find our way, hoping for some hint, some clue, of what is going on; but all we see is more and more senseless code.

Of course you have been impeded by bad code. So then—why did you write it?

Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn't have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had promised to get done and realized that you needed to slam this module together so you could move on to the next. We've all done it.

We've all looked at the mess we've just made and then have chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a

1. [Beck07].

working mess is better than nothing. We've all said we'd go back and clean it up later. Of course, in those days we didn't know LeBlanc's law: *Later equals never.*

The Total Cost of Owning a Mess

If you have been a programmer for more than two or three years, you have probably been significantly slowed down by someone else's messy code. If you have been a programmer for longer than two or three years, you have probably been slowed down by messy code. The degree of the slowdown can be significant. Over the span of a year or two, teams that were moving very fast at the beginning of a project can find themselves moving at a snail's pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial. Every addition or modification to the system requires that the tangles, twists, and knots be "understood" so that more tangles, twists, and knots can be added. Over time the mess becomes so big and so deep and so tall, they can not clean it up. There is no way at all.

As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero. As productivity decreases, management does the only thing they can; they add more staff to the project in hopes of increasing productivity. But that new staff is not versed in the design of the system. They don't know the difference between a change that matches the design intent and a change that thwarts the design intent. Furthermore, they, and everyone else on the team, are under horrific pressure to increase productivity. So they all make more and more messes, driving the productivity ever further toward zero. (See Figure 1-1.)

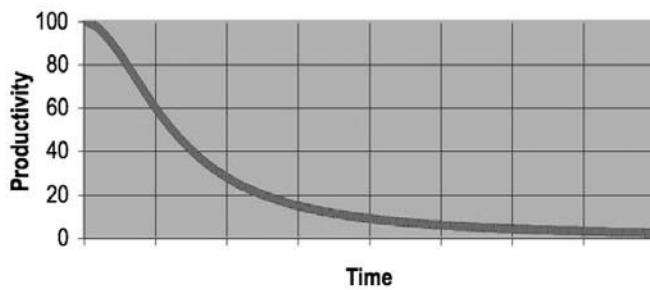


Figure 1-1
Productivity vs. time

The Grand Redesign in the Sky

Eventually the team rebels. They inform management that they cannot continue to develop in this odious code base. They demand a redesign. Management does not want to expend the resources on a whole new redesign of the project, but they cannot deny that productivity is terrible. Eventually they bend to the demands of the developers and authorize the grand redesign in the sky.

A new tiger team is selected. Everyone wants to be on this team because it's a green-field project. They get to start over and create something truly beautiful. But only the best and brightest are chosen for the tiger team. Everyone else must continue to maintain the current system.

Now the two teams are in a race. The tiger team must build a new system that does everything that the old system does. Not only that, they have to keep up with the changes that are continuously being made to the old system. Management will not replace the old system until the new system can do everything that the old system does.

This race can go on for a very long time. I've seen it take 10 years. And by the time it's done, the original members of the tiger team are long gone, and the current members are demanding that the new system be redesigned because it's such a mess.

If you have experienced even one small part of the story I just told, then you already know that spending time keeping your code clean is not just cost effective; it's a matter of professional survival.

Attitude

Have you ever waded through a mess so grave that it took weeks to do what should have taken hours? Have you seen what should have been a one-line change, made instead in hundreds of different modules? These symptoms are all too common.

Why does this happen to code? Why does good code rot so quickly into bad code? We have lots of explanations for it. We complain that the requirements changed in ways that thwart the original design. We bemoan the schedules that were too tight to do things right. We blather about stupid managers and intolerant customers and useless marketing types and telephone sanitizers. But the fault, dear Dilbert, is not in our stars, but in ourselves. We are unprofessional.

This may be a bitter pill to swallow. How could this mess be *our* fault? What about the requirements? What about the schedule? What about the stupid managers and the useless marketing types? Don't they bear some of the blame?

No. The managers and marketers look to *us* for the information they need to make promises and commitments; and even when they don't look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will fit into the system. The project managers look to us to help work out the schedule. We

are deeply complicit in the planning of the project and share a great deal of the responsibility for any failures; especially if those failures have to do with bad code!

“But wait!” you say. “If I don’t do what my manager says, I’ll be fired.” Probably not. Most managers want the truth, even when they don’t act like it. Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion; but that’s their job. It’s *your* job to defend the code with equal passion.

To drive this point home, what if you were a doctor and had a patient who demanded that you stop all the silly hand-washing in preparation for surgery because it was taking too much time?² Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of disease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.

So too it is unprofessional for programmers to bend to the will of managers who don’t understand the risks of making messes.

The Primal Conundrum

Programmers face a conundrum of basic values. All developers with more than a few years experience know that previous messes slow them down. And yet all developers feel the pressure to make messes in order to meet deadlines. In short, they don’t take the time to go fast!

True professionals know that the second part of the conundrum is wrong. You will *not* make the deadline by making the mess. Indeed, the mess will slow you down instantly, and will force you to miss the deadline. The *only* way to make the deadline—the only way to go fast—is to keep the code as clean as possible at all times.

The Art of Clean Code?

Let’s say you believe that messy code is a significant impediment. Let’s say that you accept that the only way to go fast is to keep your code clean. Then you must ask yourself: “How do I write clean code?” It’s no good trying to write clean code if you don’t know what it means for code to be clean!

The bad news is that writing clean code is a lot like painting a picture. Most of us know when a picture is painted well or badly. But being able to recognize good art from bad does not mean that we know how to paint. So too being able to recognize clean code from dirty code does not mean that we know how to write clean code!

2. When hand-washing was first recommended to physicians by Ignaz Semmelweis in 1847, it was rejected on the basis that doctors were too busy and wouldn’t have time to wash their hands between patient visits.

Writing clean code requires the disciplined use of a myriad little techniques applied through a painstakingly acquired sense of “cleanliness.” This “code-sense” is the key. Some of us are born with it. Some of us have to fight to acquire it. Not only does it let us see whether code is good or bad, but it also shows us the strategy for applying our discipline to transform bad code into clean code.

A programmer without “code-sense” can look at a messy module and recognize the mess but will have no idea what to do about it. A programmer with “code-sense” will look at a messy module and see options and variations. The “code-sense” will help that programmer choose the best variation and guide him or her to plot a sequence of behavior preserving transformations to get from here to there.

In short, a programmer who writes clean code is an artist who can take a blank screen through a series of transformations until it is an elegantly coded system.

What Is Clean Code?

There are probably as many definitions as there are programmers. So I asked some very well-known and deeply experienced programmers what they thought.

Bjarne Stroustrup, inventor of C++
and author of ***The C++ Programming
Language***

I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

Bjarne uses the word “elegant.” That’s quite a word! The dictionary in my MacBook® provides the following definitions: *pleasingly graceful and stylish in appearance or manner; pleasingly ingenious and simple*. Notice the emphasis on the word “pleasing.” Apparently Bjarne thinks that clean code is *pleasing* to read. Reading it should make you smile the way a well-crafted music box or well-designed car would.

Bjarne also mentions efficiency—*twice*. Perhaps this should not surprise us coming from the inventor of C++; but I think there’s more to it than the sheer desire for speed. Wasted cycles are inelegant, they are not pleasing. And now note the word that Bjarne uses



to describe the consequence of that inelegance. He uses the word “tempt.” There is a deep truth here. Bad code *tempts* the mess to grow! When others change bad code, they tend to make it worse.

Pragmatic Dave Thomas and Andy Hunt said this a different way. They used the metaphor of broken windows.³ A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them. They despoil the facade with graffiti and allow garbage to collect. One broken window starts the process toward decay.

Bjarne also mentions that error handing should be complete. This goes to the discipline of paying attention to details. Abbreviated error handling is just one way that programmers gloss over details. Memory leaks are another, race conditions still another. Inconsistent naming yet another. The upshot is that clean code exhibits close attention to detail.

Bjarne closes with the assertion that clean code does one thing well. It is no accident that there are so many principles of software design that can be boiled down to this simple admonition. Writer after writer has tried to communicate this thought. Bad code tries to do too much, it has muddled intent and ambiguity of purpose. Clean code is *focused*. Each function, each class, each module exposes a single-minded attitude that remains entirely undistracted, and unpolluted, by the surrounding details.

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

Grady makes some of the same points as Bjarne, but he takes a *readability* perspective. I especially like his view that clean code should read like well-written prose. Think back on a really good book that you've read. Remember how the words disappeared to be replaced by images! It was like watching a movie, wasn't it? Better! You saw the characters, you heard the sounds, you experienced the pathos and the humor.

Reading clean code will never be quite like reading *Lord of the Rings*. Still, the literary metaphor is not a bad one. Like a good novel, clean code should clearly expose the tensions in the problem to be solved. It should build those tensions to a climax and then give



3. <http://www.pragmaticprogrammer.com/booksellers/2004-12.html>

the reader that “Aha! Of course!” as the issues and tensions are resolved in the revelation of an obvious solution.

I find Grady’s use of the phrase “crisp abstraction” to be a fascinating oxymoron! After all the word “crisp” is nearly a synonym for “concrete.” My MacBook’s dictionary holds the following definition of “crisp”: *briskly decisive and matter-of-fact, without hesitation or unnecessary detail.* Despite this seeming juxtaposition of meaning, the words carry a powerful message. Our code should be matter-of-fact as opposed to speculative. It should contain only what is necessary. Our readers should perceive us to have been decisive.

**“Big” Dave Thomas, founder
of OTI, godfather of the
Eclipse strategy**

Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.

Big Dave shares Grady’s desire for readability, but with an important twist. Dave asserts that clean code makes it easy for *other* people to enhance it. This may seem obvious, but it cannot be overemphasized. There is, after all, a difference between code that is easy to read and code that is easy to change.

Dave ties cleanliness to tests! Ten years ago this would have raised a lot of eyebrows. But the discipline of Test Driven Development has made a profound impact upon our industry and has become one of our most fundamental disciplines. Dave is right. Code, without tests, is not clean. No matter how elegant it is, no matter how readable and accessible, if it hath not tests, it be unclean.

Dave uses the word *minimal* twice. Apparently he values code that is small, rather than code that is large. Indeed, this has been a common refrain throughout software literature since its inception. Smaller is better.

Dave also says that code should be *literate*. This is a soft reference to Knuth’s *literate programming*.⁴ The upshot is that the code should be composed in such a form as to make it readable by humans.



4. [Knuth92].

Michael Feathers, author of *Working Effectively with Legacy Code*

I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.

One word: care. That's really the topic of this book. Perhaps an appropriate subtitle would be *How to Care for Code*.

Michael hit it on the head. Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.

Ron Jeffries, author of *Extreme Programming Installed* and *Extreme Programming Adventures in C#*

Ron began his career programming in Fortran at the Strategic Air Command and has written code in almost every language and on almost every machine. It pays to consider his words carefully.

In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code:

- Runs all the tests;
- Contains no duplication;
- Expresses all the design ideas that are in the system;
- Minimizes the number of entities such as classes, methods, functions, and the like.

Of these, I focus mostly on duplication. When the same thing is done over and over, it's a sign that there is an idea in our mind that is not well represented in the code. I try to figure out what it is. Then I try to express that idea more clearly.

Expressiveness to me includes meaningful names, and I am likely to change the names of things several times before I settle in. With modern coding tools such as Eclipse, renaming is quite inexpensive, so it doesn't trouble me to change. Expressiveness goes



beyond names, however. I also look at whether an object or method is doing more than one thing. If it's an object, it probably needs to be broken into two or more objects. If it's a method, I will always use the Extract Method refactoring on it, resulting in one method that says more clearly what it does, and some submethods saying how it is done.

Duplication and expressiveness take me a very long way into what I consider clean code, and improving dirty code with just these two things in mind can make a huge difference. There is, however, one other thing that I'm aware of doing, which is a bit harder to explain.

After years of doing this work, it seems to me that all programs are made up of very similar elements. One example is "find things in a collection." Whether we have a database of employee records, or a hash map of keys and values, or an array of items of some kind, we often find ourselves wanting a particular item from that collection. When I find that happening, I will often wrap the particular implementation in a more abstract method or class. That gives me a couple of interesting advantages.

I can implement the functionality now with something simple, say a hash map, but since now all the references to that search are covered by my little abstraction, I can change the implementation any time I want. I can go forward quickly while preserving my ability to change later.

In addition, the collection abstraction often calls my attention to what's "really" going on, and keeps me from running down the path of implementing arbitrary collection behavior when all I really need is a few fairly simple ways of finding what I want.

Reduced duplication, high expressiveness, and early building of simple abstractions. That's what makes clean code for me.

Here, in a few short paragraphs, Ron has summarized the contents of this book. No duplication, one thing, expressiveness, tiny abstractions. Everything is there.

Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Small-talk and OO thought leader. The godfather of all those who care about code.

You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.

Statements like this are characteristic of Ward. You read it, nod your head, and then go on to the next topic. It sounds so reasonable, so obvious, that it barely registers as something profound. You might think it was pretty much what you expected. But let's take a closer look.



“... pretty much what you expected.” When was the last time you saw a module that was pretty much what you expected? Isn’t it more likely that the modules you look at will be puzzling, complicated, tangled? Isn’t misdirection the rule? Aren’t you used to flailing about trying to grab and hold the threads of reasoning that spew forth from the whole system and weave their way through the module you are reading? When was the last time you read through some code and nodded your head the way you might have nodded your head at Ward’s statement?

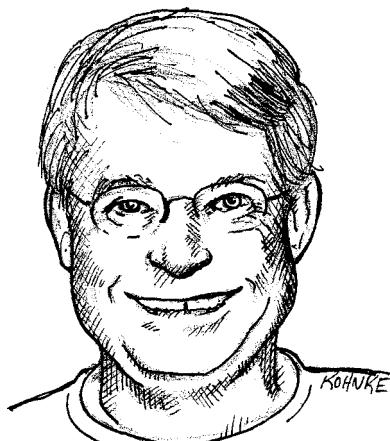
Ward expects that when you read clean code you won’t be surprised at all. Indeed, you won’t even expend much effort. You will read it, and it will be pretty much what you expected. It will be obvious, simple, and compelling. Each module will set the stage for the next. Each tells you how the next will be written. Programs that are *that* clean are so profoundly well written that you don’t even notice it. The designer makes it look ridiculously simple like all exceptional designs.

And what about Ward’s notion of beauty? We’ve all railed against the fact that our languages weren’t designed for our problems. But Ward’s statement puts the onus back on us. He says that beautiful code *makes the language look like it was made for the problem!* So it’s *our* responsibility to make the language look simple! Language bigots everywhere, beware! It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!

Schools of Thought

What about me (Uncle Bob)? What do I think clean code is? This book will tell you, in hideous detail, what I and my compatriots think about clean code. We will tell you what we think makes a clean variable name, a clean function, a clean class, etc. We will present these opinions as absolutes, and we will not apologize for our stridence. To us, at this point in our careers, they *are* absolutes. They are *our school of thought* about clean code.

Martial artists do not all agree about the best martial art, or the best technique within a martial art. Often master martial artists will form their own schools of thought and gather students to learn from them. So we see *Gracie Jiu Jitsu*, founded and taught by the Gracie family in Brazil. We see *Hakkoryu Jiu Jitsu*, founded and taught by Okuyama Ryuho in Tokyo. We see *Jeet Kune Do*, founded and taught by Bruce Lee in the United States.



Students of these approaches immerse themselves in the teachings of the founder. They dedicate themselves to learn what that particular master teaches, often to the exclusion of any other master's teaching. Later, as the students grow in their art, they may become the student of a different master so they can broaden their knowledge and practice. Some eventually go on to refine their skills, discovering new techniques and founding their own schools.

None of these different schools is absolutely *right*. Yet within a particular school we *act* as though the teachings and techniques *are* right. After all, there is a right way to practice Hakkoryu Jiu Jitsu, or Jeet Kune Do. But this rightness within a school does not invalidate the teachings of a different school.

Consider this book a description of the *Object Mentor School of Clean Code*. The techniques and teachings within are the way that *we* practice *our* art. We are willing to claim that if you follow these teachings, you will enjoy the benefits that we have enjoyed, and you will learn to write code that is clean and professional. But don't make the mistake of thinking that we are somehow "right" in any absolute sense. There are other schools and other masters that have just as much claim to professionalism as we. It would behoove you to learn from them as well.

Indeed, many of the recommendations in this book are controversial. You will probably not agree with all of them. You might violently disagree with some of them. That's fine. We can't claim final authority. On the other hand, the recommendations in this book are things that we have thought long and hard about. We have learned them through decades of experience and repeated trial and error. So whether you agree or disagree, it would be a shame if you did not see, and respect, our point of view.

We Are Authors

The @author field of a Javadoc tells us who we are. We are authors. And one thing about authors is that they have readers. Indeed, authors are *responsible* for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.

You might ask: How much is code really read? Doesn't most of the effort go into writing it?

Have you ever played back an edit session? In the 80s and 90s we had editors like Emacs that kept track of every keystroke. You could work for an hour and then play back your whole edit session like a high-speed movie. When I did this, the results were fascinating.

The vast majority of the playback was scrolling and navigating to other modules!

Bob enters the module.

He scrolls down to the function needing change.

He pauses, considering his options.

Oh, he's scrolling up to the top of the module to check the initialization of a variable.

Now he scrolls back down and begins to type.

*Ooops, he's erasing what he typed!
He types it again.
He erases it again!
He types half of something else but then erases that!
He scrolls down to another function that calls the function he's changing to see how it is called.
He scrolls back up and types the same code he just erased.
He pauses.
He erases that code again!
He pops up another window and looks at a subclass. Is that function overridden?*

...

You get the drift. Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are *constantly* reading old code as part of the effort to write new code.

Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. Of course there's no way to write code without reading it, so *making it easy to read actually makes it easier to write*.

There is no escape from this logic. You cannot write code if you cannot read the surrounding code. The code you are trying to write today will be hard or easy to write depending on how hard or easy the surrounding code is to read. So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read.

The Boy Scout Rule

It's not enough to write the code well. The code has to be *kept clean* over time. We've all seen code rot and degrade as time passes. So we must take an active role in preventing this degradation.

The Boy Scouts of America have a simple rule that we can apply to our profession.

Leave the campground cleaner than you found it.⁵

If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot. The cleanup doesn't have to be something big. Change one variable name for the better, break up one function that's a little too large, eliminate one small bit of duplication, clean up one composite `if` statement.

Can you imagine working on a project where the code *simply got better* as time passed? Do you believe that any other option is professional? Indeed, isn't continuous improvement an intrinsic part of professionalism?

5. This was adapted from Robert Stephenson Smyth Baden-Powell's farewell message to the Scouts: "Try and leave this world a little better than you found it . . ."

Prequel and Principles

In many ways this book is a “prequel” to a book I wrote in 2002 entitled *Agile Software Development: Principles, Patterns, and Practices* (PPP). The PPP book concerns itself with the principles of object-oriented design, and many of the practices used by professional developers. If you have not read PPP, then you may find that it continues the story told by this book. If you have already read it, then you’ll find many of the sentiments of that book echoed in this one at the level of code.

In this book you will find sporadic references to various principles of design. These include the Single Responsibility Principle (SRP), the Open Closed Principle (OCP), and the Dependency Inversion Principle (DIP) among others. These principles are described in depth in PPP.

Conclusion

Books on art don’t promise to make you an artist. All they can do is give you some of the tools, techniques, and thought processes that other artists have used. So too this book cannot promise to make you a good programmer. It cannot promise to give you “code-sense.” All it can do is show you the thought processes of good programmers and the tricks, techniques, and tools that they use.

Just like a book on art, this book will be full of details. There will be lots of code. You’ll see good code and you’ll see bad code. You’ll see bad code transformed into good code. You’ll see lists of heuristics, disciplines, and techniques. You’ll see example after example. After that, it’s up to you.

Remember the old joke about the concert violinist who got lost on his way to a performance? He stopped an old man on the corner and asked him how to get to Carnegie Hall. The old man looked at the violinist and the violin tucked under his arm, and said: “Practice, son. Practice!”

Bibliography

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Knuth92]: *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

This page intentionally left blank

Index

detection, 237A–238A
++ (pre- or post-increment) operator, 325A, 326A

A

aborted computation, 109A
abstract classes, 149A, 271A, 290A
ABSTRACT FACTORY pattern, 38A, 156A, 273A, 274A
abstract interfaces, 94A
abstract methods
 adding to ArgumentMarshaler, 234A–235A
 modifying, 282A
abstract terms, 95A
abstraction
 classes depending on, 150A
 code at wrong level of, 290A–291A
 descending one level at a time, 37A
 functions descending only one level of, 304A–306A
 mixing levels of, 36A–37A
 names at the appropriate level of, 311A
 separating levels of, 305A
 wrapping an implementation, 11A
abstraction levels
 raising, 290A
 separating, 305A
accessor functions, Law of Demeter and, 98A
accessors, naming, 25A
Active Records, 101A

adapted server, 185A
affinity, 84A
Agile Software Development: Principles, Patterns, Practices (PPP), 15A
algorithms
 correcting, 269A–270A
 repeating, 48A
 understanding, 297A–298A
ambiguities
 in code, 301A
 ignored tests as, 313A
amplification comments, 59A
analysis functions, 265A
“annotation form”, of AspectJ, 166A
Ant project, 76A, 77A
AOP (aspect-oriented programming), 160A, 163A
APIs. *See also* public APIs
 calling a null-returning method from, 110A
 specialized for tests, 127A
 wrapping third-party, 108A
applications
 decoupled from Spring, 164A
 decoupling from construction details, 156A
 infrastructure of, 163A
 keeping concurrency-related code separate, 181A
arbitrary structure, 303A–304A
args array, converting into a list, 231A–232A

- A**
- Args class
 - constructing, 194A
 - implementation of, 194A–200A
 - rough drafts of, 201A–212A, 26A–231A
 - ArgsException class
 - listing, 198A–200A
 - merging exceptions into, 239A–242A
 - argument(s)
 - flag, 41A
 - for a function, 40A
 - in functions, 288A
 - monadic forms of, 41A
 - reducing, 43A
 - argument lists, 43A
 - argument objects, 43A
 - argument types
 - adding, 200A, 237A
 - negative impact of, 208A
 - ArgumentMarshaler class
 - adding the skeleton of, 213A–214A
 - birth of, 212A
 - ArgumentMarshaler interface, 197A–198A
 - arrays, moving, 279A
 - art, of clean code, 6A–7A
 - artificial coupling, 293A
 - AspectJ language, 166A
 - aspect-oriented programming (AOP), 160A, 163A
 - aspects
 - in AOP, 160A–161A
 - “first-class” support for, 166A
 - assert statements, 130A–131A
 - assertEquals, 42A
 - assertions, using a set of, 111A
 - assignments, unaligned, 87A–88A
 - atomic operation, 323A–324A
 - attributes, 68A
 - authors
 - of JUnit, 252A
 - programmers as, 13A–14A
 - authorship statements, 55A
 - automated code instrumentation, 189A–190A
 - automated suite, of unit tests, 124A
- B**
- bad code, 3A–4A. *See also* dirty code; messy code
 - degrading effect of, 250A
 - example, 71A–72A
 - experience of cleaning, 250A
 - not making up for, 55A
 - bad comments, 59A–74A
 - banner, gathering functions beneath, 67A
 - base classes, 290A, 291A
 - BDUF (Big Design Up Front), 167A
 - beans, private variables manipulated, 100A–101A
 - Beck, Kent, 3A, 34A, 71A, 171A, 252A, 289A, 296A
 - behaviors, 288A–289A
 - Big Design Up Front (BDUF), 167A
 - blank lines, in code, 78A–79A
 - blocks, calling functions within, 35A
 - Booch, Grady, 8A–9A
 - boolean, passing into a function, 41A
 - boolean arguments, 194A, 288A
 - boolean map, deleting, 224A
 - boolean output, of tests, 132A
 - bound resources, 183A, 184A
 - boundaries
 - clean, 120A
 - exploring and learning, 116A
 - incorrect behavior at, 289A
 - separating known from unknown, 118A–119A
 - boundary condition errors, 269A
 - boundary conditions
 - encapsulating, 304A
 - testing, 314A
 - boundary tests, easing a migration, 118A
 - “Bowling Game”, 312A
 - Boy Scout Rule, 14A–15A, 257A
 - following, 284A
 - satisfying, 265A
 - broken windows metaphor, 8A
 - bucket brigade, 303A

BUILD-OPERATE-CHECK pattern, 127A
builds, 287A
business logic, separating from error handling, 109A
bylines, 68A
byte-manipulation libraries, 161A, 162A–163A

C

The C++ Programming Language, 7A
calculations, breaking into intermediate values, 296A
call stack, 324A
Callable interface, 326A
caller, cluttering, 104A
calling hierarchy, 106A
calls, avoiding chains of, 98A
caring, for code, 10A
Cartesian points, 42A
CAS operation, as atomic, 328A
change(s)
 isolating from, 149A–150A
 large number of very tiny, 213A
 organizing for, 147A–150A
 tests enabling, 124A
change history, deleting, 270A
check exceptions, in Java, 106A
circular wait, 337A, 338A–339A
clarification, comments as, 57A
clarity, 25A, 26A
class names, 25A
classes
 cohesion of, 140A–141A
 creating for bigger concepts, 28A–29A
 declaring instance variables, 81A
 enforcing design and business rules, 115A
 exposing internals of, 294A
 instrumenting into ConTest, 342A
 keeping small, 136A, 175A
 minimizing the number of, 176A
 naming, 25A, 138A

nonthread-safe, 328A–329A
 as nouns of a language, 49A
 organization of, 136A
 organizing to reduce risk of change, 147A
 supporting advanced concurrency design, 183A
classification, of errors, 107A
clean boundaries, 120A
clean code
 art of, 6A–7A
 described, 7A–12A
 writing, 6A–7A
clean tests, 124A–127A
cleanliness
 acquired sense of, 6A–7A
 tied to tests, 9A
cleanup, of code, 14A–15A
clever names, 26A
client, using two methods, 330A
client code, connecting to a server, 318A
client-based locking, 185A, 329A, 330A–332A
clientScheduler, 320A
client/server application, concurrency in, 317A–321A
Client/Server nonthreaded, code for, 343A–346A
client-server using threads, code changes, 346A–347A
ClientTest.java, 318A, 344A–346A
closing braces, comments on, 67A–68A
Clover, 268A, 269A
clutter
 Javadocs as, 276A
 keeping free of, 293A
code, 2A
 bad, 3A–4A
 Beck's rules of, 10A
 commented-out, 68A–69A, 287A
 dead, 292A
 explaining yourself in, 55A
 expressing yourself in, 54A

- formatting of, 76A
- implicity of, 18A–19A
- instrumenting, 188A, 342A
- jiggling, 190A
- making readable, 311A
- necessity of, 2A
- reading from top to bottom, 37A
- simplicity of, 18A, 19A
- technique for shrouding, 20A
- third-party, 114A–115A
- width of lines in, 85A–90A
 - at wrong level of abstraction, 290A–291A
- code bases, dominated by error handling, 103A**
- code changes, comments not always following, 54A**
- code completion, automatic, 20A**
- code coverage analysis, 254A–256A**
- code instrumentation, 188A–190A**
- “code sense,” 6A, 7A
- code smells, listing of, 285A–314A**
- coding standard, 299A**
- cohesion**
 - of classes, 140A–141A
 - maintaining, 141A–146A
- command line arguments, 193A–194A**
- commands, separating from queries, 45A–46A**
- comment header standard, 55A–56A**
- comment headers, replacing, 70A**
- commented-out code, 68A–69A, 287A**
- commenting style, example of bad, 71A–72A**
- comments**
 - amplifying importance of something, 59A
 - bad, 59A–74A
 - deleting, 282A
 - as failures, 54A
 - good, 55A–59A
 - heuristics on, 286A–287A
 - HTML, 69A
 - inaccurate, 54A
- informative, 56A**
- journal, 63A–64A**
- legal, 55A–56A**
- mandated, 63A**
- misleading, 63A**
- mumbling, 59A–60A**
- as a necessary evil, 53A–59A**
- noise, 64A–66A**
- not making up for bad code, 55A**
- obsolete, 286A**
- poorly written, 287A**
- proper use of, 54A**
- redundant, 60A–62A, 272A, 275A, 286A–287A**
- restating the obvious, 64A**
- separated from code, 54A**
- TODO, 58A–59A**
- too much information in, 70A**
- venting in, 65A**
- writing, 287A**
- “communication gap”, minimizing, 168A**
- Compare and Swap (CAS) operation, 327A–328A**
- ComparisonCompactor module, 252A–265A**
 - defactored, 256A–261A
 - final, 263A–265A
 - interim, 261A–263A
 - original code, 254A–256A
- compiler warnings, turning off, 289A**
- complex code, demonstrating failures in, 341A**
- complexity, managing, 139A–140A**
- computer science (CS) terms, using for names, 27A**
- concepts**
 - keeping close to each other, 80A
 - naming, 19A
 - one word per, 26A
 - separating at different levels, 290A
 - spelling similar similarly, 20A
 - vertical openness between, 78A–79A
- conceptual affinity, of code, 84A**

- concerns**
 - cross-cutting, 160A–161A
 - separating, 154A, 166A, 178A, 250A
- concrete classes**, 149A
- concrete details**, 149A
- concrete terms**, 94A
- concurrency**
 - defense principles, 180A–182A
 - issues, 190A
 - motives for adopting, 178A–179A
 - myths and misconceptions about, 179A–180A
- concurrency code**
 - compared to nonconcurrency-related code, 181A
 - focusing, 321A
- concurrent algorithms**, 179A
- concurrent applications, partition behavior**, 183A
- concurrent code**
 - breaking, 329A–333A
 - defending from problems of, 180A
 - flaws hiding in, 188A
- concurrent programming**, 180A
 - Concurrent Programming in Java: Design Principles and Patterns*, 182A, 342A
- concurrent programs**, 178A
- concurrent update problems**, 341A
- ConcurrentHashMap implementation**, 183A
- conditionals**
 - avoiding negative, 302A
 - encapsulating, 257A–25A8, 301A
- configurable data**, 306A
- configuration constants**, 306A
- consequences, warning of**, 58A
- consistency**
 - in code, 292A
 - of enums, 278A
 - in names, 40A
- consistent conventions**, 259A
- constants**
 - versus enums, 308A–309A
 - hiding, 308A
 - inheriting, 271A, 307A–308A
 - keeping at the appropriate level, 83A
 - leaving as raw numbers, 300A
 - not inheriting, 307A–308A
 - passing as symbols, 276A
 - turning into enums, 275A–276A
- construction**
 - moving all to main, 155A, 156A
 - separating with factory, 156A
 - of a system, 154A
- constructor arguments**, 157A
- constructors, overloading**, 25A
- consumer threads**, 184A
- ConTest tool**, 190A, 342A
- context**
 - adding meaningful, 27A–29A
 - not adding gratuitous, 29A–30A
 - providing with exceptions, 107A
- continuous readers**, 184A
- control variables, within loop statements**, 80A–81A
- convenient idioms**, 155A
- convention(s)**
 - following standard, 299A–300A
 - over configuration, 164A
 - structure over, 301A
 - using consistent, 259A
- convoluted code**, 175A
- copyright statements**, 55A
- cosmic-rays**. *See one-offs*
- CountDownLatch class**, 183A
- coupling**. *See also decoupling; temporal coupling; tight coupling*
 - artificial, 293A
 - hidden temporal, 302A–303A
 - lack of, 150A
- coverage patterns, testing**, 314A
- coverage tools**, 313A
- “crisp abstraction”**, 8A–9A
- cross-cutting concerns**, 160A

Cunningham, Ward, 11A–12A
cuteness, in code, 26A

D

dangling `false` argument, 294A
data
abstraction, 93A–95A
copies of, 181A–182A
encapsulation, 181A
limiting the scope of, 181A
sets processed in parallel, 179A
types, 97A, 101A
data structures. *See also structure(s)*
compared to objects, 95A, 97A
defined, 95A
interfaces representing, 94A
treating Active Records as, 101A
data transfer-objects (DTOs),
100A–101A, 160A
database normal forms, 48A
DateInterval enum, 282A–283A
DAY enumeration, 277A
DayDate class, running `SerialDate` as,
271A
DayDateFactory, 273A–274A
dead code, 288A, 292A
dead functions, 288A
deadlock, 183A, 335A–339A
deadly embrace. *See circular wait*
debugging, finding deadlocks, 336A
decision making, optimizing,
167A–168A
decisions, postponing, 168A
declarations, unaligned, 87A–88A
DECORATOR objects, 164A
DECORATOR pattern, 274A
decoupled architecture, 167A
decoupling, from construction details,
156A
decoupling strategy, concurrency as,
178A
default constructor, deleting, 276A

degradation, preventing, 14A
deletions, as the majority of changes,
250A
density, vertical in code, 79A–80A
dependencies
finding and breaking, 250A
injecting, 157A
logical, 282A
making logical physical,
298A–299A
between methods, 329A–333A
between synchronized methods,
185A
Dependency Injection (DI), 157A
Dependency Inversion Principle (DIP),
15A, 150A
dependency magnet, 47A
dependent functions, formatting,
82A–83A
derivatives
base classes depending on, 291A
base classes knowing about, 273A
of the exception class, 48A
moving set functions into, 232A,
233A–235A
pushing functionality into, 217A
description
of a class, 138A
overloading the structure of code
into, 310A
descriptive names
choosing, 309A–310A
using, 39A–40A
design(s)
of concurrent algorithms, 179A
minimally coupled, 167A
principles of, 15A
design patterns, 290A
details, paying attention to, 8A
DI (Dependency Injection), 157A
Dijkstra, Edsger, 48A
dining philosophers execution model,
184A–185A

- DIP (Dependency Inversion Principle), 15A, 150A
dirty code. See also bad code; messy code
dirty code, cleaning, 200A
dirty tests, 123A
disinformation, avoiding, 19A–20A
distance, vertical in code, 80A–84A
distinctions, making meaningful, 20A–21A
domain-specific languages (DSLs), 168A–169A
domain-specific testing language, 127A
DoubleArgumentMarshaler class, 238A
DRY principle (Don't Repeat Yourself), 181A, 289A
DTOs (data transfer objects), 100A–101A, 160A
dummy scopes, 90A
duplicate if statements, 276A
duplication
 - of code, 48A
 - in code, 289A–290A
 - eliminating, 173A–175A
 - focusing on, 10A
 - forms of, 173A, 290A
 - reduction of, 48A
 - strategies for eliminating, 48A
- dyadic argument*, 40A
dyadic functions, 42A
dynamic proxies, 161A
- E**
- e, as a variable name*, 22A
Eclipse, 26A
edit sessions, playing back, 13A–14A
efficiency, of code, 7A
EJB architecture, early as over-engineered, 167A
EJB standard, complete overhaul of, 164A
EJB2A beans, 160A
- EJB3A, Bank object rewritten in*, 165A–166A
“elegant” code, 7A
emergent design, 171A–176A
encapsulation, 136A
 - of boundary conditions, 304A
 - breaking, 106A–107A
 - of conditionals, 301A
- encodings, avoiding*, 23A–24A, 312A–313A
entity bean, 158A–160A
enum(s)
 - changing MonthConstants to, 272A
 - using, 308A–309A
- enumeration, moving*, 277A
environment, heuristics on, 287A
environment control system, 128A–129A
envying, the scope of a class, 293A
error check, hiding a side effect, 258A
Error class, 47A–48A
error code constants, 198A–200A
error codes
 - implying a class or enum, 47A–48A
 - preferring exceptions to, 46A
 - returning, 103A–104A
 - reusing old, 48A
 - separating from the Args module, 242A–250A
- error detection, pushing to the edges*, 109A
error flags, 103A–104A
error handling, 8A, 47A–48A
error messages, 107A, 250A
error processing, testing, 238A–239A
errorMessage method, 250A
errors. See also boundary condition errors; spelling errors; string comparison errors
 - classifying, 107A
- Evans, Eric*, 311A
events, 41A
exception classification, 107A

- exception clauses, 107A–108A
- exception management code, 223A
- exceptions
 - instead of return codes, 103A–105A
 - narrowing the type of, 105A–106A
 - preferring to error codes, 46A
 - providing context with, 107A
 - separating from Args, 242A–250A
 - throwing, 104A–105A, 194A
 - unchecked, 106A–107A
- execution, possible paths of, 321A–326A
- execution models, 183A–185A
- Executor framework, 326A–327A
- ExecutorClientScheduler.java, 321A
- explanation, of intent, 56A–57A
- explanatory variables, 296A–297A
- explicitness, of code, 19A
- expressive code, 295A
- expressiveness
 - in code, 10A–11A
 - ensuring, 175A–176A
- Extract Method refactoring, 11A
- Extreme Programming Adventures in C#, 10*
- Extreme Programming Installed*, 10A
- “eye-full,” code fitting into, 79A–80A

- F**

- factories, 155A–156A
- factory classes, 273A–275A
- failure
 - to express ourselves in code, 54A
 - patterns of, 314A
 - tolerating with no harm, 330A
- false argument, 294A
- fast tests, 132A
- fast-running threads, starving longer running, 183A
- fear, of renaming, 30A
- Feathers, Michael, 10A
- feature envy
 - eliminating, 293A–294A
 - smelling of, 278A
- file size, in Java, 76A
- final keywords, 276A
- F.I.R.S.T. acronym, 132A–133A
- First Law, of TDD, 122A
- FitNesse project
 - coding style for, 90A
 - file sizes, 76A, 77A
 - function in, 32A–33A
 - invoking all tests, 224A
- flag arguments, 41A, 288A
- focussed code, 8A
- foreign code. *See third-party code*
- formatting
 - horizontal, 85A–90A
 - purpose of, 76A
 - Uncle Bob’s rules, 90A–92A
 - vertical, 76A–85A
- formatting style, for a team of developers, 90A
- Fortran, forcing encodings, 23A
- Fowler, Martin, 285A, 293A
- frame, 324A
- function arguments, 40A–45A
- function call dependencies, 84A–85A
- function headers, 70A
- function signature, 45A
- functionality, placement of, 295A–296A
- functions
 - breaking into smaller, 141A–146A
 - calling within a block, 35A
 - dead, 288A
 - defining private, 292A
 - descending one level of abstraction, 304A–306A
 - doing one thing, 35A–36A, 302A
 - dyadic, 42A
 - eliminating extraneous if statements, 262A
 - establishing the temporal nature of, 260A
 - formatting dependent, 82A–83A
 - gathering beneath a banner, 67A
 - heuristics on, 288A
 - intention-revealing, 19A
 - keeping small, 175A

length of, 34A–35A
 moving, 279A
 naming, 39A, 297A
 number of arguments in, 288A
 one level of abstraction per,
 36A–37A
 in place of comments, 67A
 renaming for clarity, 258A
 rewriting for clarity, 258A–259A
 sections within, 36A
 small as better, 34A
 structured programming with, 49A
 understanding, 297A–298A
 as verbs of a language, 49A
 writing, 49A
 futures, 326A

G

Gamma, Eric, 252A
general heuristics, 288A–307A
generated byte-code, 180A
generics, improving code readability,
 115A
get functions, 218A
getBoolean function, 224A
GETFIELD instruction, 325A, 326A
getNextId method, 326A
getState function, 129A
 Gilbert, David, 267A, 268A
given-when-then convention, 130A
glitches. See **one-offs**
global setup strategy, 155A
 “God class,” 136A–137A
good comments, 55A–59A
goto statements, avoiding, 48A, 49A
grand redesign, 5A
gratuitous context, 29A–30A

H

hand-coded instrumentation, 189A
HashTable, 328A–329A
headers. *See comment headers; function headers*

heuristics

cross references of, 286A, 409A
 general, 288A–307A
 listing of, 285A–314A
hidden temporal coupling, 259A,
 302A–303A
hidden things, in a function, 44A
hiding
 implementation, 94A
 structures, 99A
hierarchy of scopes, 88A
HN. *See Hungarian Notation*
horizontal alignment, of code, 87A–88A
horizontal formatting, 85A–90A
horizontal white space, 86A
HTML, in source code, 69A
Hungarian Notation (HN), 23A–24A,
 295A
 Hunt, Andy, 8A, 289A
hybrid structures, 99A

I

if statements
 duplicate, 276A
 eliminating, 262A
if-else chain
 appearing again and again, 290A
 eliminating, 233A
ignored tests, 313A
implementation
 duplication of, 173A
 encoding, 24A
 exposing, 94A
 hiding, 94A
 wrapping an abstraction, 11A
Implementation Patterns, 3A, 296A
implicity, of code, 18A
import lists
 avoiding long, 307A
 shortening in `Serializable`, 270A
imports, as hard dependencies, 307A
imprecision, in code, 301A
inaccurate comments, 54A

- inappropriate information, in comments**, 286A
- inappropriate static methods**, 296A
- include method**, 48A
- inconsistency, in code**, 292A
- inconsistent spellings**, 20A
- incrementalism**, 212A–214A
- indent level, of a function**, 35A
- indentation, of code**, 88A–89A
- indentation rules**, 89A
- independent tests**, 132A
- information**
 - inappropriate, 286A
 - too much, 70A, 291A–292A
- informative comments**, 56A
- inheritance hierarchy**, 308A
- inobvious connection, between a comment and code**, 70A
- input arguments**, 41A
- instance variables**
 - in classes, 140A
 - declaring, 81A
 - hiding the declaration of, 81A–82A
 - passing as function arguments, 231A
 - proliferation of, 140A
- instrumented classes**, 342A
- insufficient tests**, 313A
- integer argument(s)**
 - defining, 194A
 - integrating, 224A–225A
- integer argument functionality**,
 - moving into `ArgumentMarshaler`, 215A–216A
- integer argument type, adding to `Args`**, 212A
- integers, pattern of changes for**, 220A
- IntelliJ**, 26A
- intent**
 - explaining in code, 55A
 - explanation of, 56A–57A
 - obscured, 295A
- intention-revealing function**, 19A
- intention-revealing names**, 18A–19A
- interface(s)**
 - defining local or remote, 158A–160A
 - encoding, 24A
 - implementing, 149A–150A
 - representing abstract concerns, 150A
 - turning `ArgumentMarshaler` into, 237A
 - well-defined, 291A–292A
 - writing, 119A
- internal structures, objects hiding**, 97A
- intersection, of domains**, 160A
- intuition, not relying on**, 289A
- inventor of C++**, 7A
- Inversion of Control (IoC)**, 157A
- InvocationHandler object**, 162A
- I/O bound**, 318A
- isolating, from change**, 149A–150A
- isxxxxArg methods**, 221A–222A
- iterative process, refactoring as**, 265A

J

- jar files, deploying derivatives and bases in**, 291A
- Java**
 - aspects or aspect-like mechanisms, 161A–166A
 - heuristics on, 307A–309A
 - as a wordy language, 200A
- Java 5A, improvements for concurrent development**, 182A–183A
- Java 5A Executor framework**, 320A–321A
- Java 5A VM, nonblocking solutions in**, 327A–328A
- Java AOP frameworks**, 163A–166A
- Java programmers, encoding not needed**, 24A
- Java proxies**, 161A–163A
- Java source files**, 76A–77A
- javadocs**
 - as clutter, 276A
 - in nonpublic code, 71A
 - preserving formatting in, 270A

in public APIs, 59A
 requiring for every function, 63A
java.util.concurrent package, collections
in, 182A–183A
JBoss AOP, proxies in, 163A
JCommon library, 267A
JCommon unit tests, 270A
JDepend project, 76A, 77A
JDK proxy, providing persistence support, 161A–163A
Jeffries, Ron, 10A–11A, 289A
jiggling strategies, 190A
JNDI lookups, 157A
journal comments, 63A–64A
JUnit, 34A
JUnit framework, 252A–265A
Junit project, 76A, 77A
Just-In-Time Compiler, 180A

K

keyword form, of a function name, 43A

L

L, lower-case in variable names, 20A
language design, art of programming as, 49A
languages

 appearing to be simple, 12A
 level of abstraction, 2A
 multiple in one source file, 288A
 multiples in a comment, 270A

last-in, first-out (LIFO) data structure, operand stack as, 324A

Law of Demeter, 97A–98A, 306A

LAZY INITIALIZATION/

EVALUATION idiom, 154A

LAZY-INITIALIZATION, 157A

Lea, Doug, 182A, 342A

learning tests, 116A, 118A

LeBlanc's law, 4A

legacy code, 307A

legal comments, 55A–56A

level of abstraction, 36A–37A

levels of detail, 99A
lexicon, having a consistent, 26A
lines of code
 duplicating, 173A
 width of, 85A
list(s)
 of arguments, 43A
 meaning specific to programmers, 19A
 returning a predefined immutable, 110A
literate code, 9A
literate programming, 9A
Literate Programming, 141A
livelock, 183A, 338A
local comments, 69A–70A
local variables, 324A
 declaring, 292A
 at the top of each function, 80A
lock & wait, 337A, 338A
locks, introducing, 185A
log4j package, 116A–118A
logical dependencies, 282A, 298A–299A
LOGO language, 36A
long descriptive names, 39A
long names, for long scopes, 312A
loop counters, single-letter names for, 25A

M

magic numbers

 obscuring intent, 295A
 replacing with named constants, 300A–301A
main function, moving construction to, 155A, 156A
managers, role of, 6A
mandated comments, 63A
manual control, over a serial ID, 272A

Map
 adding for ArgumentMarshaler, 221A
 methods of, 114A
maps, breaking the use of, 222A–223A

- marshalling implementation,** 214A–215A
meaningful context, 27A–29A
member variables
 `f` prefix for, 257A
 prefixing, 24A
 renaming for clarity, 259A
mental mapping, avoiding, 25A
messy code. *See also bad code; dirty code*
 total cost of owning, 4A–12A
method invocations, 324A
method names, 25A
methods
 affecting the order of execution, 188A
 calling a twin with a flag, 278A
 changing from static to instance, 280A
 of classes, 140A
 dependencies between, 329A–333A
 eliminating duplication between, 173A–174A
 minimizing assert statements in, 176A
 naming, 25A
 tests exposing bugs in, 269A
minimal code, 9A
misleading comments, 63A
misplaced responsibility, 295A–296A, 299A
MOCK OBJECT, assigning, 155A
monadic argument, 40A
monadic forms, of arguments, 41A
monads, converting dyads into, 42A
Monte Carlo testing, 341A
Month enum, 278A
MonthConstants class, 271A
multithread aware, 332A
multithread-calculation, of throughput, 335A
multithreaded code, 188A, 339A–342A
mumbling, 59A–60A
mutators, naming, 25A
mutual exclusion, 183A, 336A, 337A
- N**
- named constants, replacing magic numbers,** 300A–301A
name-length-challenged languages, 23A
names
 abstractions, appropriate level of, 311A
 changing, 40A
 choosing, 175A, 309A–310A
 of classes, 270A–271A
 clever, 26A
 descriptive, 39A–40A
 of functions, 297A
 heuristics on, 309A–313A
 importance of, 309A–310A
 intention-revealing, 18A–19A
 length of corresponding to scope, 22A–23A
 long names for long scopes, 312A
 making unambiguous, 258A
 problem domain, 27A
 pronounceable, 21A–22A
 rules for creating, 18A–30A
 searchable, 22A–23A
 shorter generally better than longer, 30A
 solution domain, 27A
 with subtle differences, 20A
unambiguous, 312A
 at the wrong level of abstraction, 271A
naming, classes, 138A
naming conventions, as inferior to structures, 301A
navigational methods, in Active Records, 101A
near bugs, testing, 314A
negative conditionals, avoiding, 302A
negatives, 258A
nested structures, 46A
Newkirk, Jim, 116A
newspaper metaphor, 77A–78A
niladic argument, 40A
no preemption, 337A

- noise**
 - comments, 64A–66A
 - scary, 66A
 - words, 21A
- nomenclature, using standard,** 311A–312A
- nonblocking solutions**, 327A–328A
- nonconcurrency-related code**, 181A
- noninformative names**, 21A
- nonlocal information**, 69A–70A
- nonpublic code, javadocs in**, 71A
- nonstatic methods, preferred to static**, 296A
- nonthreaded code, getting working first**, 187A
- nonthread-safe classes**, 328A–329A
- normal flow**, 109A
- null**
 - not passing into methods, 111A–112A
 - not returning, 109A–110A
 - passed by a caller accidentally, 111A
- null detection logic, for**
 - `ArgumentMarshaler`, 214A
- NullPointerException**, 110A, 111A
- number-series naming**, 21A

O

- Object Oriented Analysis and Design with Applications***, 8A
- object-oriented design**, 15A
- objects**
 - compared to data structures, 95A, 97A
 - compared to data types and procedures, 101A
 - copying read-only, 181A
 - defined, 95A
- obscured intent**, 295A
- obsolete comments**, 286A
- obvious behavior**, 288A–289A
- obvious code**, 12A
- “Once and only once” principle**, 289A
- “ONE SWITCH” rule**, 299A

- one thing, functions doing**, 35A–36A, 302A
- one-offs**, 180A, 187A, 191A
- OO code**, 97A
- OO design**, 139A
- Open Closed Principle (OCP)**, 15A, 38A
 - by checked exceptions, 106A
 - supporting, 149A
- operand stack**, 324A
- operating systems, threading policies**, 188A
- operators, precedence of**, 86A
- optimistic locking**, 327A
- optimizations, LAZY-EVALUATION as**, 157A
- optimizing, decision making**, 167A–168A
- orderings, calculating the possible**, 322A–323A
- organization**
 - for change, 147A–150A
 - of classes, 136A
 - managing complexity, 139A–140A
- outbound tests, exercising an interface**, 118A
- output arguments**, 41A, 288A
 - avoiding, 45A
 - need for disappearing, 45A
- outputs, arguments as**, 45A
- overhead, incurred by concurrency**, 179A
- overloading, of code with description**, 310A

P

- paperback model, as an academic model**, 27A
- parameters, taken by instructions**, 324A
- parse operation, throwing an exception**, 220A
- partitioning**, 250A
- paths of execution**, 321A–326A
- pathways, through critical sections**, 188A

pattern names, using standard, 175A
patterns
 of failure, 314A
 as one kind of standard, 311A
performance
 of a client/server pair, 318A
 concurrency improving, 179A
 of server-based locking, 333A
permutations, calculating, 323A
persistence, 160A, 161A
pessimistic locking, 327A
phraseology, in similar names, 40A
physicalizing, a dependency, 299A
Plain-Old Java Objects. *See* POJOs
platforms, running threaded code, 188A
pleasing code, 7A
pluggable thread-based code, 187A
POJO system, agility provided by, 168A
POJOs (Plain-Old Java Objects)
 creating, 187A
 implementing business logic, 162A
 separating threaded-aware code, 190A
 in Spring, 163A
 writing application domain logic, 166A
polyadic argument, 40A
polymorphic behavior, of functions, 296A
polymorphic changes, 96A–97A
polymorphism, 37A, 299A
position markers, 67A
positives
 as easier to understand, 258A
 expressing conditionals as, 302A
 of decisions, 301
 precision as the point of all naming, 30A
predicates, naming, 25A
preemption, breaking, 338A
prefixes
 for member variables, 24A
 as useless in today's environments, 312A–313A
pre-increment operator, `++`, 324A, 325A, 326A
“prequel”, this book as, 15A
principle of least surprise, 288A–289A, 295A
principles, of design, 15A
PrintPrimes program, translation into Java, 141A
private behavior, isolating, 148A–149A
private functions, 292A
private method behavior, 147A
problem domain names, 27A
procedural code, 97A
procedural shape example, 95A–96A
procedures, compared to objects, 101A
process function, repartitioning, 319A–320A
process method, I/O bound, 319A
processes, competing for resources, 184A
processor bound, code as, 318A
producer consumer execution model, 184A
producer threads, 184A
production environment, 127A–130A
productivity, decreased by messy code, 4A
professional programmer, 25A
professional review, of code, 268A
programmers
 as authors, 13A–14A
 conundrum faced by, 6A
 responsibility for messes, 5A–6A
 unprofessional, 5A–6A
programming
 defined, 2A
 structured, 48A–49A
programs, getting them to work, 201A
pronounceable names, 21A–22A
protected variables, avoiding, 80A
proxies, drawbacks of, 163A
public APIs, javadocs in, 59A
puns, avoiding, 26A–27A
PUTFIELD instruction, as atomic, 325A

Q

queries, separating from commands, 45A–46A

R

random jiggling, tests running, 190A
range, including end-point dates in, 276A

readability

- of clean tests, 124A
- of code, 76A
- Dave Thomas on, 9A
- improving using generics, 115A

readability perspective, 8A

readers

- of code, 13A–14A
- continuous, 184A

readers-writers execution model, 184A

reading

- clean code, 8A
- code from top to bottom, 37A
- versus writing, 14A

reboots, as a lock up solution, 331A

recommendations, in this book, 13A

redesign, demanded by the team, 5A

redundancy, of noise words, 21A

redundant comments, 60A–62A, 272A, 275A, 286A–287A

ReentrantLock class, 183A

refactored programs, as longer, 146A

refactoring

- Args, 212A
- code incrementally, 172A
- as an iterative process, 265A
- putting things in to take out, 233A
- test code, 127A

Refactoring (Fowler), 285A

renaming, fear of, 30A

repeatability, of concurrency bugs, 180A

repeatable tests, 132A

requirements, specifying, 2A

resetId, byte-code generated for, 324A–325A

resources

- bound, 183A
- processes competing for, 184A
- threads agreeing on a global ordering of, 338A

responsibilities

- counting in classes, 136A
- definition of, 138A
- identifying, 139A
- misplaced, 295A–296A, 299A
- splitting a program into main, 146A

return codes, using exceptions instead, 103A–105A

reuse, 174A

risk of change, reducing, 147A

robust clear code, writing, 112A

rough drafts, writing, 200A

runnable interface, 326A

run-on expressions, 295A

run-on journal entries, 63A–64A

runtime logic, separating startup from, 154A

S

safety mechanisms, overridden, 289A

scaling up, 157A–161A

scary noise, 66A

schema, of a class, 194A

schools of thought, about clean code, 12A–13A

scissors rule, in C++, 81A

scope(s)

- defined by exceptions, 105A

- dummy, 90A

- envying, 293A

- expanding and indenting, 89A

- hierarchy in a source file, 88A

- limiting for data, 181A

- names related to the length of, 22A–23A, 312A

- of shared variables, 333A

searchable names, 22A–23A

Second Law, of TDD, 122A

sections, within functions, 36A

selector arguments, avoiding, 294A–295A

- self validating tests, 132A
- Semaphore class, 183A
- semicolon, making visible, 90A
- “serial number,” *SerialDate* using, 271A
 - SerialDate* class
 - making it right, 270A–284A
 - naming of, 270A–271A
 - refactoring, 267A–284A
 - SerialDateTests* class, 268A
- serialization, 272A
- server, threads created by, 319A–321A
- server application, 317A–318A, 343A–344A
 - server code, responsibilities of, 319A
 - server-based locking, 329A
 - as preferred, 332A–333A
 - with synchronized methods, 185A
- “Servlet” model, of Web applications, 178A
- Servlets, synchronization problems, 182A
 - set functions, moving into appropriate derivatives, 232A, 233A–235A
 - setArgument, changing, 232A–233A
 - setBoolean function, 217A
 - setter methods, injecting dependencies, 157A
- setup strategy, 155A
 - SetupTeardownIncluder.java* listing, 50A–52A
- shape classes, 95A–96A
- shared data, limiting access, 181A
- shared variables
 - method updating, 328A
 - reducing the scope of, 333A
- shotgun approach, hand-coded instrumentation as, 189A
- shut-down code, 186A
- shutdowns, graceful, 186A
- side effects
 - having none, 44A
 - names describing, 313A
- Simmons, Robert, 276A
- simple code, 10A, 12A
- Simple Design, rules of, 171A–176A
- simplicity, of code, 18A, 19A
- single assert rule, 130A–131A
- single concepts, in each test function, 131A–132A
- Single Responsibility Principle (SRP), 15A, 138A–140A
 - applying, 321A
 - breaking, 155A
 - as a concurrency defense principle, 181A
 - recognizing violations of, 174A
 - server violating, 320A
 - Sql* class violating, 147A
 - supporting, 157A
 - in test classes conforming to, 172A
 - violating, 38A
- single value, ordered components of, 42A
- single-letter names, 22A, 25A
- single-thread calculation, of throughput, 334A
- SINGLETON pattern, 274A
- small classes, 136A
- Smalltalk Best Practice Patterns*, 296A
- smart programmer, 25A
- software project, maintenance of, 175A
- software systems. *See also* system(s)
 - compared to physical systems, 158A
- SOLID class design principle, 150A
- solution domain names, 27A
- source code control systems, 64A, 68A, 69A
- source files
 - compared to newspaper articles, 77A–78A
 - multiple languages in, 288A
- Sparkle program, 34A
- spawned threads, deadlocked, 186A
- special case objects, 110A
- SPECIAL CASE PATTERN, 109A
- specifications, purpose of, 2A
- spelling errors, correcting, 20A

- `SpreadsheetDateFactory`, 274A–275A
- Spring AOP**, proxies in, 163A
- Spring Framework**, 157A
- Spring model**, following EJB3, 165A
- Spring V2.5 configuration file**, 163A–164A
- spurious failures**, 187A
- `Sql` class, changing, 147A–149A
- square root**, as the iteration limit, 74A
- SRP**. *See* Single Responsibility Principle
- standard conventions**, 299A–300A
- standard nomenclature**, 175A, 311A–312A
- standards**, using wisely, 168A
- startup process**, separating from run-time logic, 154A
- starvation**, 183A, 184A, 338A
- static function**, 279A
- static import**, 308A
- static methods**, inappropriate, 296A
- The Step-down Rule*, 37A
- stories**, implementing only today's, 158A
- STRATEGY pattern**, 290A
- string arguments**, 194A, 208A–212A, 214A–225A
- string comparison errors**, 252A
- `StringBuffers`, 129A
- Stroustrup, Bjarne**, 7A–8A
- structure(s)**. *See also* data structures
- hiding, 99A
 - hybrid, 99A
 - making massive changes to, 212A
 - over convention, 301A
- structured programming**, 48A–49A
- `SuperDashboard` class, 136A–137A
- swapping**, as permutations, 323A
- switch statements**
- burying, 37A, 38A
 - considering polymorphism before, 299A
 - reasons to tolerate, 38A–39A
- `switch/case` chain, 290A
- synchronization problems**, avoiding
- with Servlets, 182A
- synchronized block**, 334A
- synchronized keyword**, 185A
- adding, 323A
 - always acquiring a lock, 328A
 - introducing a lock via, 331A
 - protecting a critical section in code, 181A
- synchronized methods**, 185A
- synchronizing**, avoiding, 182A
- synthesis functions**, 265A
- system(s)**. *See also* software systems
- file sizes of significant, 77A
 - keeping running during development, 213A
 - needing domain-specific, 168A
- system architecture**, test driving, 166A–167A
- system failures**, not ignoring one-offs, 187A
- system level**, staying clean at, 154A
- system-wide information**, in a local comment, 69A–70A

T

- tables**, moving, 275A
- target deployment platforms**, running tests on, 341A
- task swapping**, encouraging, 188A
- TDD** (Test Driven Development), 213A
- building logic, 106A
 - as fundamental discipline, 9A
 - laws of, 122A–123A
- team rules**, 90A
- teams**
- coding standard for every, 299A–300A
 - slowed by messy code, 4A
- technical names**, choosing, 27A
- technical notes**, reserving comments for, 286A

- TEMPLATE METHOD pattern**
addressing duplication, 290A
removing higher-level duplication, 174A–175A
using, 130A
- temporal coupling.** *See also coupling*
exposing, 259A–260A
hidden, 302A–303A
side effect creating, 44A
- temporary variables, explaining,** 279A–281A
- test cases**
adding to check arguments, 237A
in `ComparisonCompactor`, 252A–254A
patterns of failure, 269A, 314A
turning off, 58A
- test code, 124A, 127A**
- TEST DOUBLE, assigning, 155A**
- Test Driven Development.** *See TDD*
- test driving, architecture, 166A–167A**
- test environment, 127A–130A**
- test functions, single concepts in, 131A–132A**
- test implementation, of an interface, 150A**
- test suite**
automated, 213A
of unit tests, 124A, 268A
verifying precise behavior, 146A
- testable systems, 172A**
- test-driven development.** *See TDD*
- testing**
arguments making harder, 40A
construction logic mixed with run-time, 155A
- testing language, domain-specific, 127A**
- testNG project, 76A, 77A**
- tests**
clean, 124A–127A
cleanliness tied to, 9A
commented out for `Serializable`, 268A–270A
dirty, 123A
enabling the -ilities, 124A
fast, 132A
fast versus slow, 314A
- heuristics on, 313A–314A
ignored, 313A
independent, 132A
insufficient, 313A
keeping clean, 123A–124A
minimizing assert statements in, 130A–131A
not stopping trivial, 313A
refactoring, 126A–127A
repeatable, 132A
requiring more than one step, 287A
running, 341A
self validating, 132A
simple design running all, 172A
suite of automated, 213A
timely, 133A
writing for multithreaded code, 339A–342A
writing for threaded code, 186A–190A
writing good, 122A–123A
- Third Law, of TDD, 122A**
- third-party code**
integrating, 116A
learning, 116A
using, 114A–115A
writing tests for, 116A
- this variable, 324A**
- Thomas, Dave, 8A, 9A, 289A**
- thread(s)**
adding to a method, 322A
interfering with each other, 330A
making as independent as possible, 182A
stepping on each other, 180A, 326A
taking resources from other threads, 338A
- thread management strategy, 320A**
- thread pools, 326A**
- thread-based code, testing, 342A**
- threaded code**
making pluggable, 187A
making tunable, 187A–188A
symptoms of bugs in, 187A
testing, 186A–190A
writing in Java 5A, 182A–183A
- threading**

adding to a client/server application, 319A, 346A–347A
 problems in complex systems, 342A
thread-safe collections, 182A–183A, 329A
throughput
 causing starvation, 184A
 improving, 319A
 increasing, 333A–335A
 validating, 318A
throws clause, 106A
tiger team, 5A
tight coupling, 172A
time, taking to go fast, 6A
Time and Money project, 76A
 file sizes, 77A
timely tests, 133A
timer program, testing, 121A–122A
 “TO” keyword, 36A
TO paragraphs, 37A
TODO comments, 58A–59A
tokens, used as magic numbers, 300A
Tomcat project, 76A, 77A
tools
 ConTest tool, 190A, 342A
 coverage, 313A
 handling proxy boilerplate, 163A
 testing thread-based code, 342A
train wrecks, 98A–99A
transformations, as return values, 41A
transitive navigation, avoiding, 306A–307A
triadic argument, 40A
triads, 42A
try blocks, 105A
try/catch blocks, 46A–47, 65A–66A
try-catch-finally statement, 105A–106A
tunable threaded-based code, 187A–188A
type encoding, 24A

U

ubiquitous language, 311A–312A
unambiguous names, 312A
unchecked exceptions, 106A–107A
unencapsulated conditional, encapsulating, 257A
unit testing, isolated as difficult, 160A
unit tests, 124A, 175A, 268A
unprofessional programming, 5A–6A
uppercase C, in variable names, 20A
usability, of newspapers, 78A
use, of a system, 154A
users, handling concurrently, 179A

V

validation, of throughput, 318A
variable names, single-letter, 25A
variables
 1 based versus zero based, 261A
 declaring, 80A, 81A, 292A
 explaining temporary, 279A–281A
 explanatory, 296A–297A
 keeping private, 93A
 local, 292A, 324A
 moving to a different class, 273A
 in place of comments, 67A
 promoting to instance variables of classes, 141A
 with unclear context, 28A
venting, in comments, 65A
verbs, keywords and, 43A
Version class, 139A
versions, not deserializing across, 272A
vertical density, in code, 79A–80A
vertical distance, in code, 80A–84A
vertical formatting, 76A–85A
vertical openness, between concepts, 78A–79A
vertical ordering, in code, 84A–85A
vertical separation, 292A

W

wading, through bad code, 3A
Web containers, decoupling provided by, 178A
what, decoupling from when, 178A
white space, use of horizontal, 86A
wildcards, 307A
Working Effectively with Legacy Code, 10A
“working” programs, 201A
workmanship, 176A
wrappers, 108A
wrapping, 108A
writers, starvation of, 184A
“Writing Shy Code,” 306A

X

XML

deployment descriptors, 160A
“policy” specified configuration files, 164A

The Clean Coder



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 Overby
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

The Clean Coder

**A CODE OF CONDUCT FOR
PROFESSIONAL PROGRAMMERS**

Robert C. Martin



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

Copyright © 2011 Pearson Education, Inc.

Illustrations copyright 2011 by Jennifer Kohnke.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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.
Second printing, August 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

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 42¹ 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 Easycoder³ 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 austere 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 book¹ 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 KLOC 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 three⁴ 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 tests⁵ 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.

HHELPING **O**THERS

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 **H**ELPED

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

A

Acceptance tests
 automated, 97B–99B
 communication and, 97B
 continuous integration and,
 104B–105B
 definition of, 94B
 developer’s role in, 100B–101B
 extra work and, 99B
 GUIs and, 103B–105B
 negotiation and, 101B–102B
 passive aggression and, 101B–102B
 timing of, 99B–100B
 unit tests and, 102B–103B
 writers of, 99B–100B
Adversarial roles, 20B–23B
Affinity estimation, 140B–141B
Ambiguity, in requirements, 92B–94B
Apologies, 6B
Apprentices, 183B
Apprenticeship, 180B–184B
Arguments, in meetings, 120B–121B
Arrogance, 16B

Automated acceptance testing,
 97B–99B
Automated quality assurance, 8B
Avoidance, 125B

B

Blind alleys, 125B–126B
Bossavit, Laurent, 83B
Bowling Game, 83B
Branching, 191B
Bug counts, 197B
Business goals, 154B

C

Caffeine, 122B
Certainty, 74B
Code
 control, 189B–194B
 owned, 157B
 3B AM, 53B–54B
 worry, 54B–55B
Coding Dojo, 83B–87B
Collaboration, 14B, 151B–160B

- Collective ownership, 157B–158B
Commitment(s), 41B–46B
 control and, 44B
 discipline and, 47B–50B
 estimation and, 132B
 expectations and, 45B
 identifying, 43B–44B
 implied, 134B–135B
 importance of, 132B
 lack of, 42B–43B
 pressure and, 146B
- Communication
 acceptance tests and, 97B
 pressure and, 148B
 of requirements, 89B–94B
- Component tests
 in testing strategy, 110B–111B
 tools for, 199B–200B
- Conflict, in meetings, 120B–121B
- Continuous build, 197B–198B
- Continuous integration, 104B–105B
- Continuous learning, 13B
- Control, commitment and, 44B
- Courage, 75B–76B
- Craftsmanship, 184B
- Creative input, 59B–60B, 123B
- Crisis discipline, 147B
- Cucumber, 200B
- Customer, identification with, 15B
- CVS, 191B
- Cycle time, in test-driven development, 72B
- D**
- Deadlines
 false delivery and, 67B
 hoping and, 65B
 overtime and, 66B
 rushing and, 65B–66B
- Debugging, 60B–63B
Defect injection rate, 75B
- Demo meetings, 120B
- Design, test-driven development and, 76B–77B
- Design patterns, 12B
- Design principles, 12B
- Details, 201B–203B
- Development. *see* test driven development (TDD)
- Disagreements, in meetings, 120B–121B
- Discipline
 commitment and, 47B–50B
 crisis, 147B
- Disengagement, 64B
- Documentation, 76B
- Domain, knowledge of, 15B
- “Done,” defining, 67B, 94B–97B
- “Do no harm” approach, 5B–10B
 to function, 5B–8B
 to structure, 8B–10B
- Driving, 64B
- E**
- Eclipse, 195B–196B
- Emacs, 195B
- Employer(s)
 identification with, 15B
 programmers vs., 153B–156B
- Estimation
 affinity, 140B–141B
 anxiety, 92B
 commitment and, 132B
 definition of, 132B–133B
 law of large numbers and, 141B
 nominal, 136B
 optimistic, 135B–136B
 PERT and, 135B–138B

pessimistic, 136B
 probability and, 133B
 of tasks, 138B–141B
 trivariate, 141B
 Expectations, commitment and, 45B
 Experience, broadening, 87B

F

Failure, degrees of, 174B
 False delivery, 67B
 FitNesse, 199B–200B
 Flexibility, 9B
 Flow zone, 56B–58B
 Flying fingers, 139B
 Focus, 121B–123B
 Function, in “do no harm” approach, 5B–8B

G

Gaillet, Emmanuel, 83B
 Gelled team, 162B–164B
 Git, 191B–194B
 Goals, 20B–23B, 118B
 Graphical user interfaces (GUIs), 103B–105B
 Green Pepper, 200B
 Grenning, James, 139B
 GUIs, 103B–105B

H

Hard knocks, 179B–180B
 Help, 67B–70B
 giving, 68B
 mentoring and, 69B–70B
 pressure and, 148B–149B
 receiving, 68B–69B
 “Hope,” 42B
 Hoping, deadlines and, 65B
 Humility, 16B

I

IDE/editor, 194B
 Identification, with employer/customer, 15B
 Implied commitments, 134B–135B
 Input, creative, 59B–60B, 123B
 Integration, continuous, 104B–105B
 Integration tests
 in testing strategy, 111B–112B
 tools for, 200B–201B
 IntelliJ, 195B–196B
 Interns, 183B
 Interruptions, 57B–58B
 Issue tracking, 196B–197B
 Iteration planning meetings, 119B
 Iteration retrospective meetings, 120B

J

JBehave, 200B
 Journeymen, 182B–183B

K

Kata, 84B–85B
 Knowledge
 of domain, 15B
 minimal, 12B
 work ethic and, 11B–13B

L

Lateness, 65B–67B
 Law of large numbers, 141B
 Learning, work ethic and, 13B
 “Let’s,” 42B
 Lindstrom, Lowell, 140B
 Locking, 190B

M

Manual exploratory tests, in testing strategy, 112B–113B
 Masters, 182B
 MDA, 201B–203B

M

- Meetings
 - agenda in, 118B
 - arguments and disagreements in, 120B–121B
 - declining, 117B
 - demo, 120B
 - goals in, 118B
 - iteration planning, 119B
 - iteration retrospective, 120B
 - leaving, 118B
 - stand-up, 119B
 - time management and, 116B–121B
- Mentoring, 14B–15B, 69B–70B, 174B–180B
- Merciless refactoring, 9B
- Messes, 126B–127B, 146B
- Methods, 12B
- Model Driven Architecture (MDA), 201B–203B
- Muscle focus, 123B
- Music, 57B

N

- “Need,” 42B
- Negotiation, acceptance tests and, 101B–102B
- Nominal estimate, 136B
- Nonprofessional, 2B

O

- Open source, 87B
- Optimistic estimate, 135B–136B
- Optimistic locking, 190B
- Outcomes, best-possible, 20B–23B
- Overtime, 66B
- Owned code, 157B
- Ownership, collective, 157B–158B

P

- Pacing, 63B–64B
- Pairing, 58B, 148B–149B, 158B
- Panic, 147B–148B
- Passion, 154B
- Passive aggression, 28B–30B, 101B–102B
- People, programmers vs., 153B–158B
- Personal issues, 54B–55B
- PERT (Program Evaluation and Review Technique), 135B–138B
- Pessimistic estimate, 136B
- Pessimistic locking, 190B
- Physical activity, 123B
- Planning Poker, 139B–140B
- Practice
 - background on, 80B–83B
 - ethics, 87B
 - experience and, 87B
 - turnaround time and, 82B–83B
 - work ethic and, 13B–14B
- Precision, premature, in requirements, 91B–92B
- Preparedness, 52B–55B
- Pressure
 - avoiding, 145B–147B
 - cleanliness and, 146B
 - commitments and, 146B
 - communication and, 148B
 - handling, 147B–149B
 - help and, 148B–149B
 - messes and, 146B
 - panic and, 147B–148B
- Priority inversion, 125B
- Probability, 133B
- Professionalism, 2B
- Programmers
 - employers vs., 153B–156B
 - people vs., 153B–158B
 - programmers vs., 157B
- Proposal, project, 31B–32B

Q

- Quality assurance (QA)
 - automated, 8B
 - as bug catchers, 6B
 - as characterizers, 108B–109B
 - ideal of, as finding no problems, 108B–109B
 - problems found by, 6B–7B
 - as specifiers, 108B
 - as team member, 108B

R

- Randori, 86B–87B
- Reading, as creative input, 59B
- Recharging, 122B–123B
- Reputation, 5B
- Requirements
 - communication of, 89B–94B
 - estimation anxiety and, 92B
 - late ambiguity in, 92B–94B
 - premature precision in, 91B–92B
 - uncertainty and, 91B–92B
- Responsibility, 2B–5B
 - apologies and, 6B
 - “do no harm” approach and, 5B–10B
 - function and, 5B–8B
 - structure and, 8B–10B
 - work ethic and, 10B–16B
- RobotFX, 200B
- Roles, adversarial, 20B–23B
- Rushing, 34B–35B, 65B–66B

S

- Santana, Carlos, 83B
- “Should,” 42B
- Showers, 64B
- Simplicity, 34B
- Sleep, 122B
- Source code control, 189B–194B

Stakes, 23B–24B

Stand-up meetings, 119B

Structure

- in “do no harm” approach, 8B–10B
- flexibility and, 9B
- importance of, 8B

SVN, 191B–194B

System tests, in testing strategy, 112B

T

Task estimation, 138B–141B

Teams and teamwork, 24B–30B

- gelled, 162B–164B
- management of, 164B
- passive aggression and, 28B–30B
- preserving, 163B
- project-initiated, 163B–164B
- project owner dilemma with, 164B–165B
- trying and, 26B–28B
- velocity of, 164B

Test driven development (TDD)

- benefits of, 74B–77B
- certainty and, 74B
- courage and, 75B–76B
- cycle time in, 72B
- debut of, 71B–72B
- defect injection rate and, 75B
- definition of, 7B–8B
- design and, 76B–77B
- documentation and, 76B
- interruptions and, 58B
- three laws of, 73B–74B
- what it is not, 77B–78B

Testing

- acceptance
 - automated, 97B–99B
 - communication and, 97B
 - continuous integration and, 104B–105B

definition of, 94B
developer’s role in, 100B–101B
extra work and, 99B
GUIs and, 103B–105B
negotiation and, 101B–102B
passive aggression and,
 101B–102B
timing of, 99B–100B
unit tests and, 102B–103B
writers of, 99B–100B
automation pyramid, 109B–113B
component
 in testing strategy, 110B–111B
 tools for, 199B–200B
importance of, 7B–8B
integration
 in testing strategy, 111B–112B
 tools for, 200B–201B
manual exploratory, 112B–113B
structure and, 9B
system, 112B
unit
 acceptance tests and, 102B–103B
 in testing strategy, 110B
 tools for, 198B–199B
TextMate, 196B
Thomas, Dave, 84B
3B AM code, 53B–54B
Time, debugging, 63B
Time management
 avoidance and, 125B
 blind alleys and, 125B–126B
 examples of, 116B
 focus and, 121B–123B
 meetings and, 116B–121B
 messes and, 126B–127B
 priority inversion and, 125B
 recharging and, 122B–123B
 “tomatoes” technique for, 124B

Tiredness, 53B–54B
“Tomatoes” time management
 technique, 124B
Tools, 189B
Trivariate estimates, 141B
Turnaround time, practice
 and, 82B–83B

U

UML, 201B
Uncertainty, requirements and,
 91B–92B
Unconventional mentoring, 179B.
 see also mentoring
Unit tests
 acceptance tests and, 102B–103B
 in testing strategy, 110B
 tools for, 198B–199B

V

Vi, 194B

W

Walking away, 64B
Wasa, 85B–86B
Wideband delphi, 138B–141B
“Wish,” 42B
Work ethic, 10B–16B
 collaboration and, 14B
 continuous learning and, 13B
 knowledge and, 11B–13B
 mentoring and, 14B–15B
 practice and, 13B–14B
Worry code, 54B–55B
Writer’s block, 58B–60B

Y

“Yes”
 cost of, 30B–34B
 learning how to say, 46B–50B