

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship

Foreword by James O. Coplien

Robert C. Martin

Clean Code

Robert C. Martin Series

The mission of this series is to improve the state of the art of software craftsmanship. The books in this series are technical, pragmatic, and substantial. The authors are highly experienced craftsmen and professionals dedicated to writing about what actually works in practice, as opposed to what might work in theory. You will read about what the author has done, not what he thinks you should do. If the book is about programming, there will be lots of code. If the book is about managing, there will be lots of case studies from real projects.

These are the books that all serious practitioners will have on their bookshelves. These are the books that will be remembered for making a difference and for guiding professionals to become true craftsman.

Managing Agile Projects

Sanjiv Augustine

Agile Estimating and Planning

Mike Cohn

Working Effectively with Legacy Code

Michael C. Feathers

Agile Java™: Crafting Code with Test-Driven Development

Jeff Langr

Agile Principles, Patterns, and Practices in C#

Robert C. Martin and Micah Martin

Agile Software Development: Principles, Patterns, and Practices

Robert C. Martin

Clean Code: A Handbook of Agile Software Craftsmanship

Robert C. Martin

UML For Java™ Programmers

Robert C. Martin

Fit for Developing Software: Framework for Integrated Tests

Rick Mugridge and Ward Cunningham

Agile Software Development with SCRUM

Ken Schwaber and Mike Beedle

Extreme Software Engineering: A Hands on Approach

Daniel H. Steinberg and Daniel W. Palmer

For more information, visit informit.com/martinseries

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
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

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 Stoughton, Massachusetts.

First printing July, 2008

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

This page intentionally left blank

Contents

Foreword.....	xix
Introduction	xxv
On the Cover.....	xxix
Chapter 1: Clean Code.....	1
There Will Be Code	2
Bad Code.....	3
The Total Cost of Owning a Mess	4
The Grand Redesign in the Sky.....	5
Attitude.....	5
The Primal Conundrum.....	6
The Art of Clean Code?.....	6
What Is Clean Code?	7
Schools of Thought.....	12
We Are Authors	13
The Boy Scout Rule.....	14
Prequel and Principles.....	15
Conclusion.....	15
Bibliography.....	15
Chapter 2: Meaningful Names	17
Introduction	17
Use Intention-Revealing Names	18
Avoid Disinformation	19
Make Meaningful Distinctions.....	20
Use Pronounceable Names.....	21
Use Searchable Names	22

Avoid Encodings	23
Hungarian Notation	23
Member Prefixes.....	24
Interfaces and Implementations	24
Avoid Mental Mapping	25
Class Names	25
Method Names	25
Don't Be Cute	26
Pick One Word per Concept	26
Don't Pun	26
Use Solution Domain Names	27
Use Problem Domain Names	27
Add Meaningful Context	27
Don't Add Gratuitous Context	29
Final Words	30
Chapter 3: Functions	31
Small!	34
Blocks and Indenting.....	35
Do One Thing	35
Sections within Functions	36
One Level of Abstraction per Function	36
Reading Code from Top to Bottom: <i>The Stepdown Rule</i>	37
Switch Statements	37
Use Descriptive Names	39
Function Arguments	40
Common Monadic Forms.....	41
Flag Arguments	41
Dyadic Functions.....	42
Triads.....	42
Argument Objects.....	43
Argument Lists	43
Verbs and Keywords	43
Have No Side Effects	44
Output Arguments	45
Command Query Separation	45

Prefer Exceptions to Returning Error Codes	46
Extract Try/Catch Blocks	46
Error Handling Is One Thing.....	47
The <code>Error.java</code> Dependency Magnet	47
Don't Repeat Yourself	48
Structured Programming	48
How Do You Write Functions Like This?	49
Conclusion	49
<code>SetupTeardownIncluder</code>	50
Bibliography	52
Chapter 4: Comments	53
Comments Do Not Make Up for Bad Code	55
Explain Yourself in Code	55
Good Comments	55
Legal Comments.....	55
Informative Comments.....	56
Explanation of Intent.....	56
Clarification.....	57
Warning of Consequences.....	58
TODO Comments.....	58
Amplification.....	59
Javadocs in Public APIs.....	59
Bad Comments	59
Mumbling	59
Redundant Comments	60
Misleading Comments.....	63
Mandated Comments.....	63
Journal Comments.....	63
Noise Comments	64
Scary Noise	66
Don't Use a Comment When You Can Use a Function or a Variable.....	67
Position Markers.....	67
Closing Brace Comments.....	67
Attributions and Bylines.....	68

Commented-Out Code.....	68
HTML Comments	69
Nonlocal Information	69
Too Much Information	70
Inobvious Connection.....	70
Function Headers.....	70
Javadocs in Nonpublic Code	71
Example.....	71
Bibliography	74
Chapter 5: Formatting	75
The Purpose of Formatting	76
Vertical Formatting	76
The Newspaper Metaphor	77
Vertical Openness Between Concepts	78
Vertical Density	79
Vertical Distance	80
Vertical Ordering.....	84
Horizontal Formatting	85
Horizontal Openness and Density	86
Horizontal Alignment.....	87
Indentation.....	88
Dummy Scopes.....	90
Team Rules	90
Uncle Bob’s Formatting Rules	90
Chapter 6: Objects and Data Structures	93
Data Abstraction	93
Data/Object Anti-Symmetry	95
The Law of Demeter	97
Train Wrecks	98
Hybrids	99
Hiding Structure	99
Data Transfer Objects	100
Active Record.....	101
Conclusion	101
Bibliography	101

Chapter 7: Error Handling	103
Use Exceptions Rather Than Return Codes	104
Write Your Try-Catch-Finally Statement First	105
Use Unchecked Exceptions	106
Provide Context with Exceptions	107
Define Exception Classes in Terms of a Caller's Needs	107
Define the Normal Flow	109
Don't Return Null	110
Don't Pass Null	111
Conclusion	112
Bibliography	112
Chapter 8: Boundaries	113
Using Third-Party Code	114
Exploring and Learning Boundaries	116
Learning log4j	116
Learning Tests Are Better Than Free	118
Using Code That Does Not Yet Exist	118
Clean Boundaries	120
Bibliography	120
Chapter 9: Unit Tests	121
The Three Laws of TDD	122
Keeping Tests Clean	123
Tests Enable the -ilities.....	124
Clean Tests	124
Domain-Specific Testing Language.....	127
A Dual Standard	127
One Assert per Test	130
Single Concept per Test.....	131
F.I.R.S.T.	132
Conclusion	133
Bibliography	133
Chapter 10: Classes	135
Class Organization	136
Encapsulation	136

Classes Should Be Small!	136
The Single Responsibility Principle.....	138
Cohesion.....	140
Maintaining Cohesion Results in Many Small Classes.....	141
Organizing for Change	147
Isolating from Change.....	149
Bibliography	151
Chapter 11: Systems	153
How Would You Build a City?	154
Separate Constructing a System from Using It	154
Separation of Main	155
Factories	155
Dependency Injection.....	157
Scaling Up	157
Cross-Cutting Concerns	160
Java Proxies	161
Pure Java AOP Frameworks	163
AspectJ Aspects	166
Test Drive the System Architecture	166
Optimize Decision Making	167
Use Standards Wisely, When They Add <i>Demonstrable Value</i>	168
Systems Need Domain-Specific Languages	168
Conclusion	169
Bibliography	169
Chapter 12: Emergence	171
Getting Clean via Emergent Design	171
Simple Design Rule 1: Runs All the Tests	172
Simple Design Rules 2–4: Refactoring	172
No Duplication	173
Expressive	175
Minimal Classes and Methods	176
Conclusion	176
Bibliography	176
Chapter 13: Concurrency	177
Why Concurrency?	178
Myths and Misconceptions.....	179

Challenges	180
Concurrency Defense Principles	180
Single Responsibility Principle	181
Corollary: Limit the Scope of Data	181
Corollary: Use Copies of Data	181
Corollary: Threads Should Be as Independent as Possible	182
Know Your Library	182
Thread-Safe Collections	182
Know Your Execution Models	183
Producer-Consumer	184
Readers-Writers	184
Dining Philosophers	184
Beware Dependencies Between Synchronized Methods	185
Keep Synchronized Sections Small	185
Writing Correct Shut-Down Code Is Hard	186
Testing Threaded Code	186
Treat Spurious Failures as Candidate Threading Issues	187
Get Your Nonthreaded Code Working First	187
Make Your Threaded Code Pluggable	187
Make Your Threaded Code Tunable	187
Run with More Threads Than Processors	188
Run on Different Platforms	188
Instrument Your Code to Try and Force Failures	188
Hand-Coded	189
Automated	189
Conclusion	190
Bibliography	191
Chapter 14: Successive Refinement	193
Args Implementation	194
How Did I Do This?	200
Args: The Rough Draft	201
So I Stopped	212
On Incrementalism	212
String Arguments	214
Conclusion	250

Chapter 15: JUnit Internals	251
The JUnit Framework	252
Conclusion	265
Chapter 16: Refactoring SerialDate	267
First, Make It Work	268
Then Make It Right	270
Conclusion	284
Bibliography	284
Chapter 17: Smells and Heuristics	285
Comments	286
C1: <i>Inappropriate Information</i>	286
C2: <i>Obsolete Comment</i>	286
C3: <i>Redundant Comment</i>	286
C4: <i>Poorly Written Comment</i>	287
C5: <i>Commented-Out Code</i>	287
Environment	287
E1: <i>Build Requires More Than One Step</i>	287
E2: <i>Tests Require More Than One Step</i>	287
Functions	288
F1: <i>Too Many Arguments</i>	288
F2: <i>Output Arguments</i>	288
F3: <i>Flag Arguments</i>	288
F4: <i>Dead Function</i>	288
General	288
G1: <i>Multiple Languages in One Source File</i>	288
G2: <i>Obvious Behavior Is Unimplemented</i>	288
G3: <i>Incorrect Behavior at the Boundaries</i>	289
G4: <i>Overridden Safeties</i>	289
G5: <i>Duplication</i>	289
G6: <i>Code at Wrong Level of Abstraction</i>	290
G7: <i>Base Classes Depending on Their Derivatives</i>	291
G8: <i>Too Much Information</i>	291
G9: <i>Dead Code</i>	292
G10: <i>Vertical Separation</i>	292
G11: <i>Inconsistency</i>	292
G12: <i>Clutter</i>	293

G13: <i>Artificial Coupling</i>	293
G14: <i>Feature Envy</i>	293
G15: <i>Selector Arguments</i>	294
G16: <i>Obscured Intent</i>	295
G17: <i>Misplaced Responsibility</i>	295
G18: <i>Inappropriate Static</i>	296
G19: <i>Use Explanatory Variables</i>	296
G20: <i>Function Names Should Say What They Do</i>	297
G21: <i>Understand the Algorithm</i>	297
G22: <i>Make Logical Dependencies Physical</i>	298
G23: <i>Prefer Polymorphism to If/Else or Switch/Case</i>	299
G24: <i>Follow Standard Conventions</i>	299
G25: <i>Replace Magic Numbers with Named Constants</i>	300
G26: <i>Be Precise</i>	301
G27: <i>Structure over Convention</i>	301
G28: <i>Encapsulate Conditionals</i>	301
G29: <i>Avoid Negative Conditionals</i>	302
G30: <i>Functions Should Do One Thing</i>	302
G31: <i>Hidden Temporal Couplings</i>	302
G32: <i>Don't Be Arbitrary</i>	303
G33: <i>Encapsulate Boundary Conditions</i>	304
G34: <i>Functions Should Descend Only One Level of Abstraction</i>	304
G35: <i>Keep Configurable Data at High Levels</i>	306
G36: <i>Avoid Transitive Navigation</i>	306
Java	307
J1: <i>Avoid Long Import Lists by Using Wildcards</i>	307
J2: <i>Don't Inherit Constants</i>	307
J3: <i>Constants versus Enums</i>	308
Names	309
N1: <i>Choose Descriptive Names</i>	309
N2: <i>Choose Names at the Appropriate Level of Abstraction</i>	311
N3: <i>Use Standard Nomenclature Where Possible</i>	311
N4: <i>Unambiguous Names</i>	312
N5: <i>Use Long Names for Long Scopes</i>	312
N6: <i>Avoid Encodings</i>	312
N7: <i>Names Should Describe Side-Effects</i>	313

Tests	313
T1: <i>Insufficient Tests</i>	313
T2: <i>Use a Coverage Tool!</i>	313
T3: <i>Don't Skip Trivial Tests</i>	313
T4: <i>An Ignored Test Is a Question about an Ambiguity</i>	313
T5: <i>Test Boundary Conditions</i>	314
T6: <i>Exhaustively Test Near Bugs</i>	314
T7: <i>Patterns of Failure Are Revealing</i>	314
T8: <i>Test Coverage Patterns Can Be Revealing</i>	314
T9: <i>Tests Should Be Fast</i>	314
Conclusion	314
Bibliography	315
Appendix A: Concurrency II	317
Client/Server Example	317
The Server	317
Adding Threading	319
Server Observations	319
Conclusion	321
Possible Paths of Execution	321
Number of Paths	322
Digging Deeper	323
Conclusion	326
Knowing Your Library	326
Executor Framework	326
Nonblocking Solutions	327
Nonthread-Safe Classes	328
Dependencies Between Methods	
Can Break Concurrent Code	329
Tolerate the Failure	330
Client-Based Locking	330
Server-Based Locking	332
Increasing Throughput	333
Single-Thread Calculation of Throughput	334
Multithread Calculation of Throughput	335
Deadlock	335
Mutual Exclusion	336
Lock & Wait	337

No Preemption.....	337
Circular Wait	337
Breaking Mutual Exclusion.....	337
Breaking Lock & Wait.....	338
Breaking Preemption.....	338
Breaking Circular Wait.....	338
Testing Multithreaded Code	339
Tool Support for Testing Thread-Based Code	342
Conclusion	342
Tutorial: Full Code Examples	343
Client/Server Nonthreaded	343
Client/Server Using Threads	346
Appendix B: org.jfree.date.SerialDate	349
Appendix C: Cross References of Heuristics	409
Epilogue	411
Index	413

This page intentionally left blank

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.
- *Shutsuke*, 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.

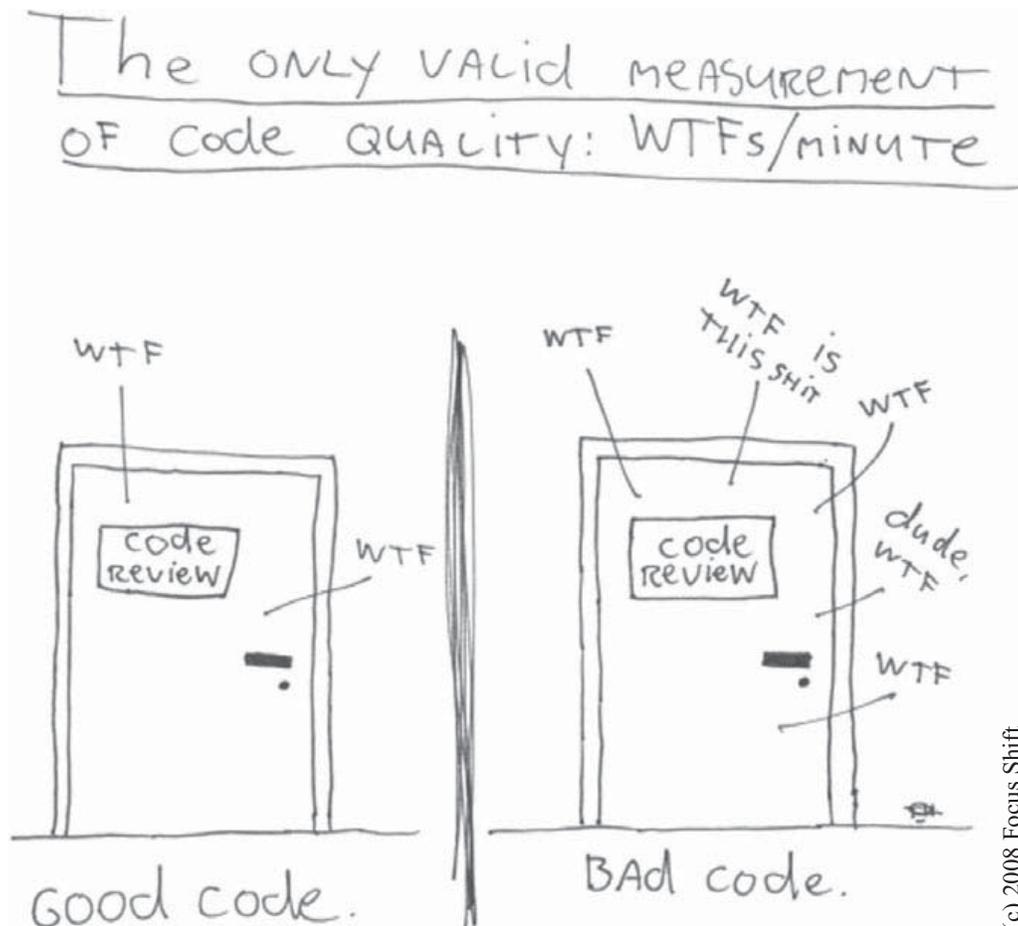
In our profession, we desperately need all the help we can get. If a clean shop floor reduces accidents, and well-organized shop tools increase productivity, then I'm all for them. As for this book, it is the best pragmatic application of Lean principles to software I have ever seen in print. I expected 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



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

Artwork

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.

This page intentionally left blank

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