Robert C. Martin Series

Clean Agile

Back to Basics

With Contributions from Jerry Fitzpatrick, Tim Ottinger, Jeff Langr, Eric Crichlow, Damon Poole, *and* Sandro Mancuso

Robert C. Martin

Praise for Clean Agile

"In the journey to all things Agile, Uncle Bob has been there, done that, and has both the t-shirt and the scars to show for it. This delightful book is part history, part personal stories, and all wisdom. If you want to understand what Agile is and how it came to be, this is the book for you."

—Grady Booch

"Bob's frustration colors every sentence of *Clean Agile*, but it's a justified frustration. What *is* in the world of Agile development is nothing compared to what *could be*. This book is Bob's perspective on what to focus on to get to that 'what could be.' And he's been there, so it's worth listening."

—Kent Beck

"It's good to read Uncle Bob's take on Agile. Whether just beginning, or a seasoned *agilista*, you would do well to read this book. I agree with almost all of it. It's just some of the parts make me realize my own shortcomings, darn it. It made me double-check our code coverage (85.09%)."

—Jon Kern

"This book provides a historical lens through which to view Agile development more fully and accurately. Uncle Bob is one of the smartest people I know, and he has boundless enthusiasm for programming. If anyone can demystify Agile development, it's him."

—From the Foreword by Jerry Fitzpatrick

This page intentionally left blank

Clean Agile

Clean Agile

Back to Basics

Robert C. Martin

Pearson

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

Visit us on the Web: informit.com

Library of Congress Control Number: 2019945397

Cover image: Peresanz/Shutterstock

Foreword by Jerry Fitzpatrick, Software Renovation Corporation, March 2019. Used with permission.

Chapter 7 by Sandro Mancuso, April 27, 2019. Used with permission.

Afterword by Eric Crichlow, April 5, 2019. Used with permission.

ISBN-13: 978-0-13-578186-9
ISBN-10: 0-13-578186-8

ScoutAutomatedPrintLine

To every programmer who ever tilted at windmills or waterfalls.

This page intentionally left blank

Contents

Foreword

What exactly is Agile development? How did it originate? How has it evolved?

In this book, Uncle Bob provides thoughtful answers to these questions. He also identifies the many ways in which Agile development has been misinterpreted or corrupted. His perspective is relevant because he is an authority on the subject, having participated in the birth of Agile development.

Bob and I have been friends for many years. We first met when I joined the telecommunications division of Teradyne in 1979. As an electrical engineer, I helped install and support products; later, I became a hardware designer.

About a year after I joined, the company began seeking new product ideas. In 1981, Bob and I proposed an electronic telephone receptionist—essentially a voicemail system with call-routing features. The company liked the concept, and we soon began developing "E.R.—The Electronic Receptionist." Our prototype was state of the art. It ran the MP/M operating system on an Intel 8086 processor. Voice messages were stored on a five-megabyte Seagate ST-506 hard disk. I designed the voice port hardware while Bob started writing the application. When I finished my design, I wrote application code, too, and I have been a developer ever since.

Around 1985 or 1986, Teradyne abruptly halted E.R. development and, unknown to us, withdrew the patent application. It was a business decision that the company would soon regret, and one that still haunts Bob and me.

Eventually, each of us left Teradyne for other opportunities. Bob started a consulting business in the Chicago area. I became a software contractor and instructor. We managed to stay in touch even though I moved to another state.

By the year 2000, I was teaching Object-Oriented Analysis and Design for Learning Tree International. The course incorporated UML and the Unified Software Development Process (USDP). I was well versed in these technologies, but not with Scrum, Extreme Programming, or similar methodologies.

In February 2001, the Agile Manifesto was published. Like many developers, my initial reaction was "The Agile what?" The only manifesto I knew of was from Karl Marx, an avid Communist. Was this Agile thing a call to arms? Dang software radicals!

The Manifesto did start a rebellion of sorts. It was meant to inspire the development of lean, clean code by using a collaborative, adaptive, feedback-driven approach. It offered an alternative to "heavyweight" processes like Waterfall and the USDP.

It has been 18 years since the Agile Manifesto was published. It is, therefore, ancient history to most of today's developers. For this reason, your understanding of Agile development may not line up with the intent of its creators.

This book aims to set the record straight. It provides a historical lens through which to view Agile development more fully and accurately. Uncle Bob is one of the smartest people I know, and he has boundless enthusiasm for programming. If anyone can demystify Agile development, it's him.

—Jerry Fitzpatrick
Software Renovation Corporation
March 2019

Preface

This book is not a work of research. I have not done a diligent literature review. What you are about to read are my personal recollections, observations, and opinions about my 20-year involvement with Agile—nothing more, nothing less.

The writing style is conversational and colloquial. My word choices are sometimes a bit crude. And though I am not one to swear, one [slightly modified] curse word made it into these pages because I could think of no better way to convey the intended meaning.

Oh, this book isn't a complete rave. When it struck me as necessary, I cited some references for you to follow. I checked some of my facts against those of other folks who've been in the Agile community as long as I have. I've even asked several folks to provide supplemental and disagreeing points of view in their own chapters and sections. Still, you should not think of this book as a scholarly work. It may be better to think of it as a memoir—the grumblings of a curmudgeon telling all those new-fangled Agile kids to get off his lawn.

This book is for programmers and non-programmers alike. It is not technical. There is no code. It is meant to provide an overview of the original intent of Agile software development without getting into any deep technical details of programming, testing, and managing.

This is a small book. That's because the topic isn't very big. Agile is a small idea about the small problem of small programming teams doing small things. Agile is *not* a big idea about the big problem of big programming teams doing big things. It's somewhat ironic that this small solution to a small problem has a name. After all, the small problem in question was solved in the 1950s and '60s, almost as soon as software was invented. Back in those days, small software teams learned to do small things rather well. However, it all got derailed in the 1970s when the small software teams doing small things got all tangled up in an ideology that thought it should be doing big things with big teams.

Aren't we supposed to be doing big things with big teams? Heavens, no! Big things don't get done by big teams; big things get done by the collaboration

of many small teams doing many small things. This is what the programmers in the 1950s and ’60s knew instinctively. And it was this that was forgotten in the 1970s.

Why was this forgotten? I suspect it was because of a discontinuity. The number of programmers in the world began to explode in the 1970s. Prior to that, there were only a few thousand programmers in the world. After that, there were hundreds of thousands. Now that number is approaching one hundred million.

Those first programmers back in the 1950s and ’60s were not youngsters. They started programming in their 30s, 40s, and 50s. By the 1970s, just when the population of programmers was staring to explode, those oldsters were starting to retire. So the necessary training never occurred. An impossibly young cohort of 20-somethings entered the workforce just as the experienced folks were leaving, and their experience was not effectively transferred.

Some would say that this event started a kind of dark ages in programming. For 30 years, we struggled with the idea that we should be doing big things with big teams, never knowing that the secret was to do many small things with many small teams.

Then in the mid ’90s, we began to realize what we had lost. The idea of small teams began to germinate and grow. The idea spread through the community of software developers, gathering steam. In 2000, we realized we needed an industry-wide reboot. We needed to be reminded of what our forebears instinctively knew. We needed, once again, to realize that big things are done by many collaborating small teams doing small things.

To help popularize this, we gave the idea a name. We called it “Agile.”

I wrote this preface in the first days of 2019. It’s been nearly two decades since the reboot of 2000, and it seems to me that it’s time for yet another. Why? Because the simple and small message of Agile has become muddled over the intervening years. It’s been mixed with the concepts of Lean,

Kanban, LeSS, SAFe, Modern, Skilled, and so many others. These other ideas are not bad, but they are not the original Agile message.

So it's time, once again, for us to be reminded of what our forebears knew in the '50s and '60s, and what we relearned in 2000. It's time to remember what Agile really is.

In this book, you will find nothing particularly new, nothing astounding or startling, nothing revolutionary that breaks the mold. What you will find is a restatement of Agile as it was told in 2000. Oh, it's told from a different perspective, and we have learned a few things over the last 20 years that I'll include. But overall, the message of this book is the message of 2001 and the message of 1950.

It's an old message. It's a true message. It's a message that gives us the small solution to the small problem of small software teams doing small things.

Acknowledgments

My first acknowledgment goes to a pair of intrepid programmers who joyously discovered (or rediscovered) the practices contained herein: Ward Cunningham and Kent Beck.

Next in line is Martin Fowler, without whose steadying hand, in those earliest of days, the Agile revolution would likely have been stillborn.

Ken Schwaber deserves a special mention for the indomitable energy he applied toward the promotion and adoption of Agile.

Mary Poppendieck also deserves special mention for the selfless and inexhaustible energy she put into the Agile movement and her shepherding of the Agile Alliance.

In my view, Ron Jeffries, through his talks, articles, blogs, and the persistent warmth of his character, acted as the conscience of the early Agile movement.

Mike Beedle fought the good fight for Agile but was senselessly murdered by a homeless person on the streets of Chicago.

The other original authors of the Agile Manifesto take a special place here:

Arie van Bennekum, Alistair Cockburn, James Grenning, Jim Highsmith, Andrew Hunt, Jon Kern, Brian Marick, Steve Mellor, Jeff Sutherland, and Dave Thomas.

Jim Newkirk, my friend and business partner at the time, worked tirelessly in support of Agile while enduring personal headwinds that most of us (and certainly I) can't begin to imagine.

Next, I'd like to mention the folks who worked at Object Mentor Inc. They all took the initial risk of adopting and promoting Agile. Many of them are in the following photo, taken at the kickoff of the first XP Immersion course.

Back Row: Ron Jeffries, author, Brian Button, Lowell Lindstrom, Kent Beck, Micah Martin, Angelique Martin, Susan Rosso, James Grenning.
Front Row: David Farber, Eric Meade, Mike Hill, Chris Biegay, Alan Francis, Jennifer Kohnke, Talisha Jefferson, Pascal Roy.
Not pictured: Tim Ottinger, Jeff Langr, Bob Koss, Jim Newkirk, Michael Feathers, Dean Wampler, and David Chelimsky.

I'd also like to acknowledge the folks who gathered to form the Agile Alliance. Some of them are in the picture that follows, which was taken at the kickoff meeting of that now-august alliance.

Left to right: Mary Poppendieck, Ken Schwaber, author, Mike Beedle, Jim Highsmith. (Not pictured: Ron Crocker.)

Finally, thanks to all the folks at Pearson, especially my publisher Julie Phifer.

This page intentionally left blank

About the Author

Robert C. Martin (**Uncle Bob**) has been a programmer since 1970. He is co-founder of cleancoders.com, offering on-line video training for software developers, and founder of Uncle Bob Consulting LLC, offering software consulting, training, and skill development services to major corporations worldwide. He served as the Master Craftsman at 8th Light Inc., a Chicago-based software consulting firm.

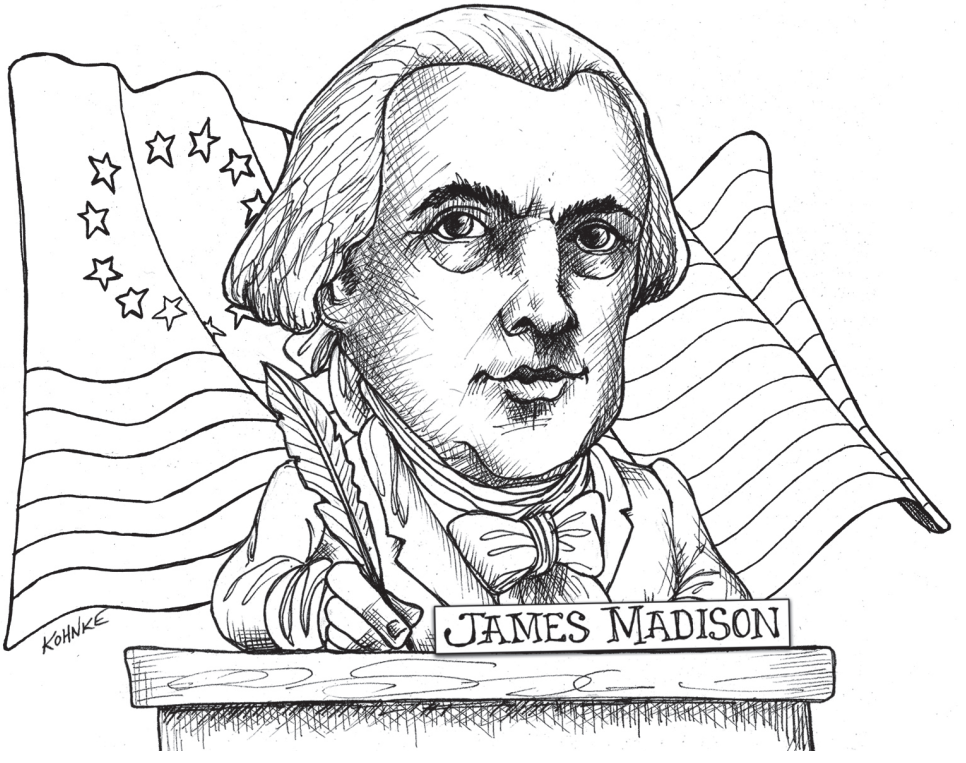
Mr. Martin has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He is also the creator of the acclaimed educational video series at cleancoders.com.
Mr. Martin has authored and edited many books including the following:

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

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

A leader in the industry of software development, Mr. Martin served three years as the editor-in-chief of the C++ *Report*, and he served as the first chairman of the Agile Alliance.

2 The Reasons for Agile



Before we dive into the details of Agile development, I want to explain what's at stake. Agile development is important, not just to software development, but to our industry, our society, and ultimately our civilization.

Developers and managers are often attracted to Agile development for transient reasons. They might try it because it just somehow feels right to them, or perhaps they fell for the promises of speed and quality. These reasons are intangible, indistinct, and easily thwarted. Many people have dropped Agile development simply because they didn't immediately experience the outcome they thought were promised.

These evanescent reasons are not why Agile development is important. Agile development is important for much deeper philosophical and ethical reasons. Those reasons have to do with professionalism and the reasonable expectations of our customers.

Professionalism

What drew me to Agile in the first place was the high commitment to discipline over ceremony. To do Agile right, you had to work in pairs, write tests first, refactor, and commit to simple designs. You had to work in short cycles, producing executable output in each. You had to communicate with business on a regular and continuous basis.

Look back at the Circle of Life and view each one of those practices as a *promise,* a *commitment,* and you'll see where I am coming from. For me, Agile development is a commitment to up my game—to be a professional, and to promote professional behavior throughout the industry of software development.

We in this industry sorely need to increase our professionalism. We fail too often. We ship too much crap. We accept too many defects. We make terrible trade-offs. Too often, we behave like unruly teenagers with a new credit card. In simpler times, these behaviors were tolerable because the stakes were relatively low. In the '70s and '80s and even into the '90s, the cost of software failure, though high, was limited and containable.

SOFTWARE IS EVERYWHERE

Today things are different.

Look around you, right now. Just sit where you are and look around the room. How many computers are in the room with you?

Here, let me do that. Right now, I am at my summer cabin in the north woods of Wisconsin. How many computers are in this room with me?

- 4: I'm writing this on a MacBook Pro with 4 cores. I know, they say 8, but I don't count "virtual" cores. I also won't count all the little ancillary processors in the MacBook.
- 1: My Apple Magic Mouse 2. I'm sure it has more than one processor in it, but I'll just count it as 1.
- 1: My iPad running Duet as a second monitor. I know there are lots of other little processors in the iPad, but I'll only count it as one.
- 1: My car key (!).
- 3: My Apple AirPods. One for each earpiece, and one for the case. There are probably more in there but…
- 1: My iPhone. Yeah, yeah, the real number of processors in the iPhone is probably above a dozen, but I'll keep it at one.
- 1: Ultrasonic motion detector in sight. (There are many more in the house, but only one that I can see.)
- 1: Thermostat.
- 1: Security panel.
- 1: Flat-screen TV.
- 1: DVD player.
- 1: Roku Internet TV streaming device.
- 1: Apple AirPort Express.
- 1: Apple TV.
- 5: Remote controls.
- 1: Telephone. (Yes, an actual telephone.)
- 1: Fake fireplace. (You should see all the fancy modes it's got!)

- 2: Old computer-controlled telescope, a Meade LX 200 EMC. One processor in the drive and another in the handheld control unit.
- 1: Thumb drive in my pocket.
- 1: Apple pencil.

I count at least 30 computers on my person and in this room with me. The real number is probably double that since most of the devices have multiple processors in them. But let's just stick with 30 for the moment.

What did you count? I'll bet that for most of you it came close to my 30. Indeed, I'll wager that most of the 1.3 billion people living in Western society are constantly near more than a dozen computers. That's new. In the early '90s, that number would have averaged closer to zero.

What do every single one of those nearby computers have in common? They all need to be programmed. They all need software—software written by us. And what, do you think, is the quality of that software?

Let me put this in a different light. How many times per day does your grandmother interact with a software system? For those of you who still have living grandmothers that number will likely be in the thousands, because in today's society you can't do anything without interacting with a software system. You can't

- Talk on the phone.
- Buy or sell anything.
- Use the microwave oven, refrigerator, or even the toaster.
- Wash or dry your clothes.
- Wash the dishes.
- Listen to music.
- Drive a car.
- File an insurance claim.
- Increase the temperature in the room.
- Watch TV.

But it's worse than that. Nowadays, in our society, virtually nothing of significance can be done without interacting with a software system. No law can be passed, enacted, or enforced. No government policy can be debated. No plane can be flown. No car can be driven. No missile launched. No ship sailed. Roads can't be paved, food can't be harvested, steel mills can't mill steel, auto factories can't make cars, candy companies can't make candy, stocks can't be traded…

Nothing gets done in our society without software. Every waking moment is dominated by software. Many of us even monitor our sleep with software.

WE RULE THE WORLD

Our society has become utterly and wholly dependent on software. Software is the life's blood that makes our society run. Without it, the civilization we currently enjoy would be impossible.

And who writes all that software? You and I. We, programmers, rule the world.

Other people think they rule the world, but then they hand the rules they've made to us and *we* write the actual rules that run in the machines that monitor and control virtually every activity of modern life.

We, programmers, rule the world.

And we are doing a pretty poor job of it.

How much of that software, that runs absolutely everything, do you think is properly tested? How many programmers can say that they have a test suite that *proves,* with a high degree of certainty, that the software they have written works?

Do the hundred million lines of code that run inside your car work? Have you found any bugs in it? I have. What about the code that controls the brakes, and the accelerator, and the steering? Any bugs in that? Is there a test suite that can

be run at a moment's notice that *proves* with a high degree of certainty that when you put your foot on the brake pedal, the car will actually stop?

How many people have been killed because the software in their cars failed to heed the pressure of the driver's foot on the brake pedal? We don't know for sure, but the answer is *many*. In one 2013 case Toyota paid millions in damages because the software contained "possible bit flips, task deaths that would disable the fail-safes, memory corruption, single-point failures, inadequate protections against stack overflow and buffer overflow, single-fault containment regions, [and] thousands of global variables" all within "spaghetti code."[1]

Our software is now killing people. You and I probably didn't get into this business to kill people. Many of us are programmers because, as kids, we wrote an infinite loop that printed our name on the screen, and we just thought that was so cool. But now our actions are putting lives and fortunes at stake. And with every passing day, more and more code puts more and more lives and fortunes at stake.

The Disaster

The day will come, if it hasn't already by the time you read this, when some poor programmer is going to do some dumb thing and kill ten thousand people in a single moment of carelessness. Think about that for a minute. It's not hard to imagine half a dozen scenarios. And when that happens, the politicians of the world will rise up in righteous indignation (as they should) and point their fingers squarely at us.

You might think that those fingers would point at our bosses, or the executives in our companies, but we saw what happened when those fingers pointed to the CEO of Volkswagen, North America, as he testified before Congress. The politicians asked him why Volkswagen had put software in their cars that purposely detected and defeated the emissions testing hardware

1. Safety Research & Strategies Inc. 2013. Toyota unintended acceleration and the big bowl of "spaghetti" code [blog post]. November 7. Accessed at http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code.

used in California. He replied, "This was not a corporate decision, from my point of view, and to my best knowledge today. This was a couple of software engineers who put this in for whatever reasons."[2]

So, those fingers will point at us. And rightly so. Because it will have been our fingers on the keyboards, our disciplines that were lacking, and our carelessness that was the ultimate cause.

It was with this in mind that I held such high hopes for Agile. I hoped then, as I hope today, that the disciplines of Agile software development would be our first step toward turning computer programming into a true and honorable profession.

Reasonable Expectations

What follows is a list of perfectly reasonable expectations that managers, users, and customers have of us. Notice as you read through this list that one side of your brain agrees that each item is perfectly reasonable. Notice that the other side of your brain, the programmer side, reacts in horror. The programmer side of your brain may not be able to imagine how to meet these expectations.

Meeting these expectations is one of the primary goals of Agile development. The principles and practices of Agile address most of the expectations on this list quite directly. The behaviors below are what any good chief technology officer (CTO) should expect from their staff. Indeed, to drive this point home, I want you to think of me as your CTO. Here is what I expect.

We Will Not Ship Shyt!

It is an unfortunate aspect of our industry that this expectation even has to be mentioned. But it does. I'm sure, dear readers, that many of you have fallen afoul of this expectation on one or more occasions. I certainly have.

2. O'Kane, S. 2015. Volkswagen America's CEO blames software engineers for emissions cheating scandal. *The Verge*. October 8. Accessed at https://www.theverge.com/2015/10/8/9481651/volkswagen-congressional-hearing-diesel-scandal-fault.

To understand just how severe this problem is, consider the shutdown of the Air Traffic Control network over Los Angeles due to the rollover of a 32-bit clock. Or the shutdown of all the power generators on board the Boeing 787 for the same reason. Or the hundreds of people killed by the 737 Max MCAS software.

Or how about my own experience with the early days of healthcare.gov? After initial login, like so many systems nowadays, it asked for a set of security questions. One of those was "A memorable date." I entered `7/21/73`, my wedding anniversary. The system responded with `Invalid Entry`.

I'm a programmer. I know how programmers think. So I tried many different date formats: `07/21/1973`, `07-21-1973`, `21 July, 1973`, `07211973`, etc. All gave me the same result. `Invalid Entry`. This was frustrating. What date format did the blasted thing want?

Then it occurred to me. The programmer who wrote this didn't know what questions would be asked. He or she was just pulling the questions from a database and storing the answers. That programmer was probably also disallowing special characters and numbers in those answers. So I typed: `Wedding Anniversary`. This was accepted.

I think it's fair to say that any system that requires its users to think like programmers in order to enter data in the expected format is crap.

I could fill this section with anecdotes about crappy software like this. But others have done this far better than I could. If you want to get a much better idea of the scope of this issue, read Gojko Adzic's book *Humans vs. Computers*[3] and Matt Parker's *Humble Pi*.[4]

It is perfectly reasonable for our managers, customers, and users to expect that we will provide systems for them that are high in quality and low in

3. Adzic, G. 2017. *Humans vs. Computers*. London: Neuri Consulting LLP. Accessed at http://humansvscomputers.com.
4. Parker, M. 2019. *Humble Pi: A Comedy of Maths Errors*. London: Penguin Random House UK. Accessed at https://mathsgear.co.uk/products/humble-pi-a-comedy-of-maths-errors.

defect. Nobody expects to be handed crap—especially when they pay good money for it.

Note that Agile's emphasis on Testing, Refactoring, Simple Design, and customer feedback is the obvious remedy for shipping bad code.

Continuous Technical Readiness

The last thing that customers and managers expect is that we, programmers, will create artificial delays to shipping the system. But such artificial delays are common in software teams. The cause of such delays is often the attempt to build all features simultaneously instead of the most important features first. So long as there are features that are half done, or half tested, or half documented, the system cannot be deployed.

Another source of artificial delays is the notion of stabilization. Teams will frequently set aside a period of continuous testing during which they watch the system to see if it fails. If no failures are detected after X days, the developers feel safe to recommend the system for deployment.

Agile resolves these issues with the simple rule that the system should be *technically* deployable at the end of every iteration. Technically deployable means that from the developers' points of view the system is technically solid enough to be deployed. The code is clean and the tests all pass.

This means that the work completed in the iteration includes all the coding, all the testing, all the documentation, and all the stabilization for the stories implemented in that iteration.

If the system is *technically* ready to deploy at the end of every iteration, then deployment is a *business decision,* not a technical decision. The business may decide there aren't enough features to deploy, or they may decide to delay deployment for market reasons or training reasons. In any case, the system quality meets the *technical* bar for deployability.

Is it possible for the system to be technically deployable every week or two? Of course it is. The team simply has to pick a batch of stories that is small enough to allow them to complete all the deployment readiness tasks before the end of the iteration. They'd better be automating the vast majority of their testing, too.

From the point of view of the business and the customers, continuous technical readiness is simply expected. When the business sees a feature work, they expect that feature is done. They don't expect to be told that they have to wait a month for QA stabilization. They don't expect that the feature only worked because the programmers driving the demo bypassed all the parts that don't work.

Stable Productivity

You may have noticed that programming teams can often go very fast in the first few months of a greenfield project. When there's no existing code base to slow you down, you can get a lot of code working in a short time.

Unfortunately, as time passes, the messes in the code can accumulate. If that code is not kept clean and orderly, it will put a back pressure on the team that slows progress. The bigger the mess, the higher the back pressure, and the slower the team. The slower the team, the greater the schedule pressure, and the greater the incentive to make an even bigger mess. That positive-feedback loop can drive a team to near immobility.

Managers, puzzled by this slowdown, may finally decide to add human resources to the team in order to increase productivity. But as we saw in the previous chapter, adding personnel actually slows down the team for a few weeks.

The hope is that after those weeks the new people will come up to speed and help to increase the velocity. But who is training the new people? The people who made the mess in the first place. The new people will certainly emulate that established behavior.

Worse, the existing code is an even more powerful instructor. The new people will look at the old code and surmise how things are done in this team, and they will continue the practice of making messes. So the productivity continues to plummet despite the addition of the new folks.

Management might try this a few more times because repeating the same thing and expecting different results is the definition of management sanity in some organizations. In the end, however, the truth will be clear. Nothing that managers do will stop the inexorable plunge towards immobility.

In desperation, the managers ask the developers what can be done to increase productivity. And the developers have an answer. They have known for some time what needs to be done. They were just waiting to be asked.

"Redesign the system from scratch." The developers say.

Imagine the horror of the managers. Imagine the money and time that has been invested so far into this system. And now the developers are recommending that the whole thing be thrown away and redesigned from scratch!

Do those managers believe the developers when they promise, "This time things will be different"? Of course they don't. They'd have to be fools to believe that. Yet, what choice do they have? Productivity is on the floor. The business isn't sustainable at this rate. So, after much wailing and gnashing of teeth, they agree to the redesign.

A cheer goes up from the developers. "Hallelujah! We are all going back to the beginning when life is good and code is clean!" Of course, that's not what happens at all. What really happens is that the team is split in two. The ten best, The Tiger Team—the guys who made the mess in the first place—are chosen and moved into a new room. They will lead the rest of us into the golden land of a redesigned system. The rest of us hate those guys because now we're stuck maintaining the old crap.

From where does the Tiger Team get their requirements? Is there an up-to-date requirements document? Yes. It's the old code. The old code is the only document that accurately describes what the redesigned system should do.

So now the Tiger Team is poring over the old code trying to figure out just what it does and what the new design ought to be. Meanwhile the rest of us are changing that old code, fixing bugs and adding new features.

Thus, we are in a race. The Tiger Team is trying to hit a moving target. And, as Zeno showed in the parable of Achilles and the tortoise, trying to catch up to a moving target can be a challenge. Every time the Tiger Team gets to where the old system was, the old system has moved on to a new position.

It requires calculus to prove that Achilles will eventually pass the tortoise. In software, however, that math doesn't always work. I worked at a company where ten years later the new system had not yet been deployed. The customers had been promised a new system eight years before. But the new system never had enough features for those customers; the old system always did more than the new system. So the customers refused to take the new system.

After a few years, customers simply ignored the promise of the new system. From their point of view that system didn't exist, and it never would.

Meanwhile, the company was paying for two development teams: the Tiger Team and the maintenance team. Eventually, management got so frustrated that they told their customers they were deploying the new system despite their objections. The customers threw a fit over this, but it was nothing compared to the fit thrown by the developers on the Tiger Team—or, should I say, the remnants of the Tiger Team. The original developers had all been promoted and gone off to management positions. The current members of the team stood up in unison and said, "You can't ship this, it's crap. It needs to be redesigned."

OK, yes, another hyperbolic story told by Uncle Bob. The story is based on truth, but I did embellish it for effect. Still, the underlying message is entirely true. Big redesigns are horrifically expensive and seldom are deployed.

Customers and managers don't expect software teams to slow down with time. Rather, they expect that a feature similar to one that took two weeks at the start of a project will take two weeks a year later. They expect productivity to be stable over time.

Developers should expect no less. By continuously keeping the architecture, design, and code as clean as possible, they can keep their productivity high and prevent the otherwise inevitable spiral into low productivity and redesign.

As we will show, the Agile practices of Testing, Pairing, Refactoring, and Simple Design are the technical keys to breaking that spiral. And the Planning Game is the antidote to the schedule pressure that drives that spiral.

Inexpensive Adaptability

Software is a compound word. The word "ware" means "product." The word "soft" means easy to change. Therefore, software is a product that is easy to change. Software was invented because we wanted a way to quickly and easily change the behavior of our machines. Had we wanted that behavior to be hard to change, we would have called it hardware.

Developers often complain about changing requirements. I have often heard statements like, "That change completely thwarts our architecture." I've got some news for you, sunshine. If a change to the requirements breaks your architecture, then your architecture sucks.

We developers should celebrate change because that's why we are here. Changing requirements is the name of the whole game. Those changes are the justification for our careers and our salaries. Our jobs depend on our ability to accept and engineer changing requirements and to make those changes relatively inexpensive.

To the extent that a team's software is hard to change, that team has thwarted the very reason for that software's existence. Customers, users, and managers expect that software systems will be easy to change and that the cost of such changes will be small and proportionate.

We will show how the Agile practices of TDD, Refactoring, and Simple Design all work together to make sure that software systems can be safely changed with a minimum of effort.

CONTINUOUS IMPROVEMENT

Humans make things better with time. Painters improve their paintings, songwriters improve their songs, and homeowners improve their homes. The same should be true for software. The older a software system is, the *better* it should be.

The design and architecture of a software system should get better with time. The structure of the code should improve, and so should the efficiency and throughput of the system. Isn't that obvious? Isn't that what you would expect from any group of humans working on anything?

It is the single greatest indictment of the software industry, the most obvious evidence of our failure as professionals, that we make things worse with time. The fact that we developers expect our systems to get messier, cruftier, and more brittle and fragile with time is, perhaps, the most irresponsible attitude possible.

Continuous, steady improvement is what users, customers, and managers expect. They expect that early problems will fade away and that the system will get better and better with time. The Agile practices of Pairing, TDD, Refactoring, and Simple Design strongly support this expectation.

FEARLESS COMPETENCE

Why don't most software systems improve with time? Fear. More specifically, fear of change.

Imagine you are looking at some old code on your screen. Your first thought is, "This is ugly code, I should clean it." Your next thought is, "I'm not

touching it!" Because you know if you touch it, you will break it; and if you break it, it will become yours. So you back away from the one thing that might improve the code: cleaning it.

This is a fearful reaction. You fear the code, and this fear forces you to behave incompetently. You are incompetent to do the necessary cleaning because you fear the outcome. You have allowed this code, which you created, to go so far out of your control that you fear any action to improve it. This is irresponsible in the extreme.

Customers, users, and managers expect *fearless competence*. They expect that if you see something wrong or dirty, you will fix and clean it. They don't expect you to allow problems to fester and grow; they expect you to stay on top of the code, keeping it as clean and clear as possible.

So how do you eliminate that fear? Imagine that you own a button that controls two lights: one red, the other green. Imagine that when you push this button, the green light is lit if the system works, and the red light is lit if the system is broken. Imagine that pushing that button and getting the result takes just a few seconds. How often would you push that button? You'd never stop. You'd push that button all the time. Whenever you made any change to the code, you'd push that button to make sure you hadn't broken anything.

Now imagine that you are looking at some ugly code on your screen. Your first thought is, "I should clean it." And then you simply start to clean it, pushing the button after each small change to make sure you haven't broken anything.

The fear is gone. You can clean the code. You can use the Agile practices of Refactoring, Pairing, and Simple Design to improve the system.

How do you get that button? The Agile practice of TDD provides that button for you. If you follow that practice with discipline and determination, you will have that button, and you will be fearlessly competent.

QA SHOULD FIND NOTHING

QA should find no faults with the system. When QA runs their tests, they should come back saying that everything works as required. Any time QA finds a problem, the development team should find out what went wrong in their process and fix it so that next time QA will find nothing.

QA should wonder why they are stuck at the back end of the process checking systems that always work. And, as we shall see, there is a much better place for QA to be positioned.

The Agile practices of Acceptance Tests, TDD, and Continuous Integration support this expectation.

TEST AUTOMATION

The hands you see in the picture in Figure 2.1 are the hands of a QA manager. The document that manager is holding is *the table of contents* for a *manual* test plan. It lists 80,000 manual tests that must be run every six months by an army of testers in India. It costs over a million dollars to run those tests.

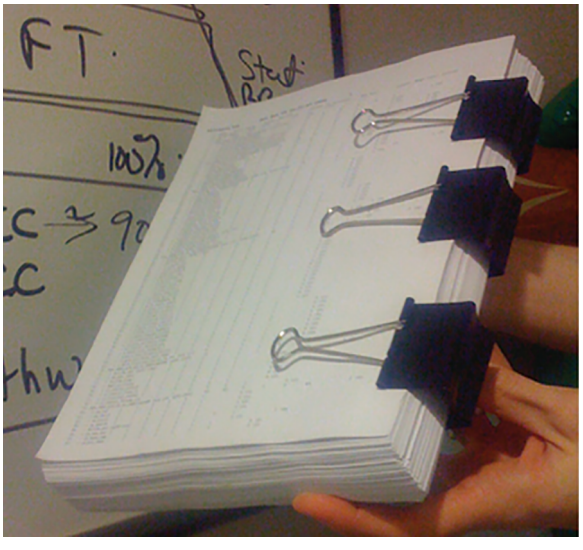


Figure 2.1 Table of contents for the manual test plan

The QA manager is holding this document out to me because he just got back from his boss' office. His boss just got back from the CFO's office. The year is 2008. The great recession has begun. The CFO cut that million dollars in half every six months. The QA manager is holding this document out to me asking me which half of these tests he shouldn't run.

I told him that no matter how he decided to cut the tests, he wouldn't know if half of his system was working.

This is the inevitable result of manual testing. Manual tests are always eventually lost. What you just heard was the first and most obvious mechanism for losing manual tests: manual tests are *expensive* and so are always a target for reduction.

However, there is a more insidious mechanism for losing manual tests. Developers seldom deliver to QA on time. This means that QA has less time than planned to run the tests they need to run. So, QA must *choose* which tests they believe are most appropriate to run in order to make the shipment deadline. And so some tests aren't run. They are lost.

And besides, humans are not machines. Asking humans to do what machines can do is expensive, inefficient, and *immoral*. There is a much better activity for which QA should be employed—an activity that uses their human creativity and imagination. But we'll get to that.

Customers and users expect that every new release is thoroughly tested. No one expects the development team to bypass tests just because they ran out of time or money. So every test that can feasibly be automated must be automated. Manual testing should be limited to those things that cannot be automatically validated and to the creative discipline of Exploratory Testing.[5]

The Agile practices of TDD, Continuous Integration, and Acceptance Testing support this expectation.

5. Agile Alliance. Exploratory testing. Accessed at https://www.agilealliance.org/glossary/exploratory-testing.

We Cover for Each Other

As CTO, I expect development teams to behave like teams. How do teams behave? Imagine a team of players moving the ball down the field. One of the players trips and falls. What do the other players do? They cover the open hole left behind by the fallen team member *and continue to move the ball down the field*.

On board a ship, everyone has a job. Everyone also knows how to do someone else's job. Because on board the ship, all jobs must get done.

In a software team, if Bob gets sick, Jill steps in to finish Bob's tasks. This means that Jill had better know what Bob was working on and where Bob keeps all the source files, and scripts, etc.

I expect that the members of each software team will cover for each other. I expect that each individual member of a software team makes sure that there is someone who can cover for him if he goes down. It is *your* responsibility to make sure that one or more of your teammates can cover for you.

If Bob is the database guy, and Bob gets sick, I don't expect progress on the project to grind to a halt. Someone else, even though she isn't "the database guy," should pick up the slack. I don't expect the team to keep knowledge in silos; I expect knowledge to be shared. If I need to reassign half the members of the team to a new project, I do not expect that half the knowledge will be removed from the team.

The Agile practices of Pair Programming, Whole Team, and Collective Ownership support these expectations.

Honest Estimates

I expect estimates, and I expect them to be honest. The most honest estimate is "I don't know." However, that estimate is not complete. You may not know everything, but there are some things you do know. So I expect you to provide estimates based on what you do *and don't* know.

For example, you may not know how long something will take, but you can compare one task to another in relative terms. You may not know how long it will take to build the *Login* page, but you might be able to tell me that the *Change Password* page will take about half the time as *Login*. Relative estimates like that are immensely valuable, as we will see in a later chapter.

Or, instead of estimating in relative terms, you may be able to give me a range of probabilities. For example, you might tell me that the *Login* page will take anywhere from 5 to 15 days to complete with an average completion time of 12 days. Such estimates combine what you do *and don't* know into an honest probability for managers to manage.

The Agile practices of the Planning Game and Whole Team support this expectation.

You Need to Say "No"

While it is important to strive to find solutions to problems, I expect you to say "no" when no such solution can be found. You need to realize that you were hired more for your ability to say "no" than for your ability to code. You, programmers, are the ones who know whether something is possible. As your CTO, I am counting on you to inform us when we are headed off a cliff. I expect that, no matter how much schedule pressure you feel, no matter how many managers are demanding results, you will say "no" when the answer really is "no."

The Agile practice of Whole Team supports this expectation.

Continuous Aggressive Learning

As CTO, I expect you to keep learning. Our industry changes quickly. We must be able to change with it. So learn, learn, learn! Sometimes the company can afford to send you to courses and conferences. Sometimes the company can afford to buy books and training videos. But if not, then you must find ways to continue learning without the company's help.

The Agile practice of Whole Team supports this expectation.

Mentoring

As CTO I expect you to teach. Indeed, the best way to learn is to teach. So when new people join the team, teach them. Learn to teach each other.

Again, the Agile practice of Whole Team supports this expectation.

The Bill of Rights

During the Snowbird meeting, Kent Beck said that the goal of Agile was to heal the divide between business and development. To that end, the following bill of rights was developed by Kent, Ward Cunningham, and Ron Jeffries, among others.

Notice, as you read these rights, that the rights of the customer and the rights of the developer are complementary. They fit together like a hand in a glove. They create a balance of expectations between the two groups.

Customer Bill of Rights

The customer bill of rights includes the following:

- You have the right to an overall plan and to know what can be accomplished when and at what cost.
- You have the right to get the most possible value out of every iteration.
- You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify.
- You have the right to change your mind, to substitute functionality, and to change priorities without paying exorbitant costs.
- You have the right to be informed of schedule and estimate changes, in time to choose how to reduce the scope to meet a required date. You can cancel at any time and be left with a useful working system reflecting investment to date.

Developer Bill of Rights

The developer bill of rights includes the following:

- You have the right to know what is needed with clear declarations of priority.
- You have the right to produce high-quality work at all times.
- You have the right to ask for and receive help from peers, managers, and customers.
- You have the right to make and update your own estimates.
- You have the right to accept your responsibilities instead of having them assigned to you.

These are extremely powerful statements. We should consider each in turn.

Customers

The word "customer" in this context refers to businesspeople in general. This includes true customers, managers, executives, project leaders, and anyone else who might carry responsibility for schedule and budget or who will pay for and benefit from the execution of the system.

> *Customers have the right to an overall plan and to know what can be accomplished when and at what cost.*

Many people have claimed that up-front planning is not part of Agile development. The very first customer right belies that claim. Of course the business needs a plan. Of course that plan must include schedule and cost. And, of course that plan should be as accurate and precise as practical.

It is in that last clause that we often get into trouble because the only way to be both accurate and precise is to actually develop the project. Being both accurate and precise by doing anything less is impossible. So what we developers must do to guarantee this right is to make sure that our plans, estimates, and schedules properly describe the level of our uncertainty and define the means by which that uncertainty can be mitigated.

In short, we cannot agree to deliver fixed scopes on hard dates. Either the scopes or the dates must be soft. We represent that softness with probability curve. For example, we estimate that there is a 95% probability that we can get the first ten stories done by the date. A 50% chance that we can get the next five done by the date. And a 5% chance that the five after that might get done by the date.

Customers have the right to this kind of probability-based plan because they cannot manage their business without it.

Customers have the right to get the most possible value out of every iteration.

Agile breaks up the development effort into fixed time boxes called *iterations*. The business has the right to expect that the developers will work on the most important things at any given time, and that each iteration will provide them the maximum possible *usable* business value. This priority of value is specified by the customer during the planning sessions at the beginning of each iteration. The customers choose the stories that give them the highest return on investment and that can fit within the developer's estimation for the iteration.

Customers have the right to see progress in a running system, proven to work by passing repeatable tests that they specify.

This seems obvious when you think about it from the customer's point of view. Of course they have the right to see incremental progress. Of course they have the right to specify the criteria for accepting that progress. Of course they have the right to quickly and repeatedly see proof that their acceptance criteria have been met.

Customers have the right to change their minds, to substitute functionality, and to change priorities without paying exorbitant costs.

After all, this is *soft*ware. The whole point of software is to be able to easily change the behavior of our machines. The softness is the reason software was

invented in the first place. So of course, customers have the right to change the requirements.

> *Customers have the right to be informed of schedule and estimate changes in time to choose how to alter the scope to meet the required date.*

> *Customers may cancel at any time and be left with a useful working system reflecting investment to date.*

Note that customers do not have the right to demand conformance to the schedule. Their right is limited to managing the schedule by changing the scope. The most important thing this right confers is the right to *know* that the schedule is in jeopardy so that it can be managed in a timely fashion.

Developers

In this context, developers are anyone who works on the development of code. This includes programmers, QA, testers, and business analysts.

> *Developers have the right to know what is needed with clear declarations of priority.*

Again, the focus is on *knowledge*. Developers are entitled to precision in the requirements and in the importance of those requirements. Of course, the same constraint of *practicality* holds for requirements as holds for estimates. It is not always possible to be perfectly precise about requirements. And indeed, customers have the right to change their minds.

So this right only applies *within the context of an iteration*. Outside of an iteration, requirements and priorities will shift and change. But within an iteration the developers have the right to consider them immutable. Always remember, however, that developers may choose to waive that right if they consider a requested change to be inconsequential.

> *Developers have the right to produce high-quality work at all times.*

This may be the most profound of all these rights. Developers have the right to do good work. The business has no right to tell developers to cut corners or do low-quality work. Or, to say this differently, the business has no right to force developers to ruin their professional reputations or violate their professional ethics.

Developers have the right to ask for and receive help from peers, managers, and customers.

This help comes in many forms. Programmers may ask each other for help solving a problem, checking a result, or learning a framework, among other things. Developers might ask customers to better explain requirements or to refine priorities. Mostly, this statement gives programmers the right to *communicate*. And with that right to ask for help comes the responsibility to give help when asked.

Developers have the right to make and update their own estimates.

No one can estimate a task for you. And if you estimate a task, you can always change your estimate when new factors come to light. Estimates are guesses. They are intelligent guesses to be sure, but they're still guesses. They are guesses that get better with time. Estimates are never commitments.

Developers have the right to accept their responsibilities instead of having them assigned.

Professionals *accept* work, they are not assigned work. A professional developer has every right to say "no" to a particular job or task. It may be that the developer does not feel confident in their ability to complete the task, or it may be that the developer believes the task better suited for someone else. Or, it may be that the developer rejects the tasks for personal or moral reasons.[6]

6. Consider the developers at Volkswagen who "accepted" the tasks of cheating the EPA test rigs in California. https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal.

In any case, the right to accept comes with a cost. Acceptance implies responsibility. The accepting developer becomes responsible for the quality and execution of the task, for continually updating the estimate so that the schedule can be managed, for communicating status to the whole team, and for asking for help when help is needed.

Programming on a team involves working closely together with juniors and seniors. The team has the right to collaboratively decide who will do what. A technical leader might ask a developer to take a task but has no right to force a task on anyone.

Conclusion

Agile is a framework of disciplines that support professional software development. Those who profess these disciplines accept and conform to the reasonable expectations of managers, stakeholders, and customers. They also enjoy and abide by the rights that Agile confers to developers and customers. This mutual acceptance and conference of rights and expectations—this profession of disciplines—is the foundation of an *ethical* standard for software.

Agile is not a process. Agile is not a fad. Agile is not merely a set of rules. Rather, Agile is a set of rights, expectations, and disciplines of the kind that form the basis of an ethical profession.

Index

D

E

Q

R

S

T

U

V

W