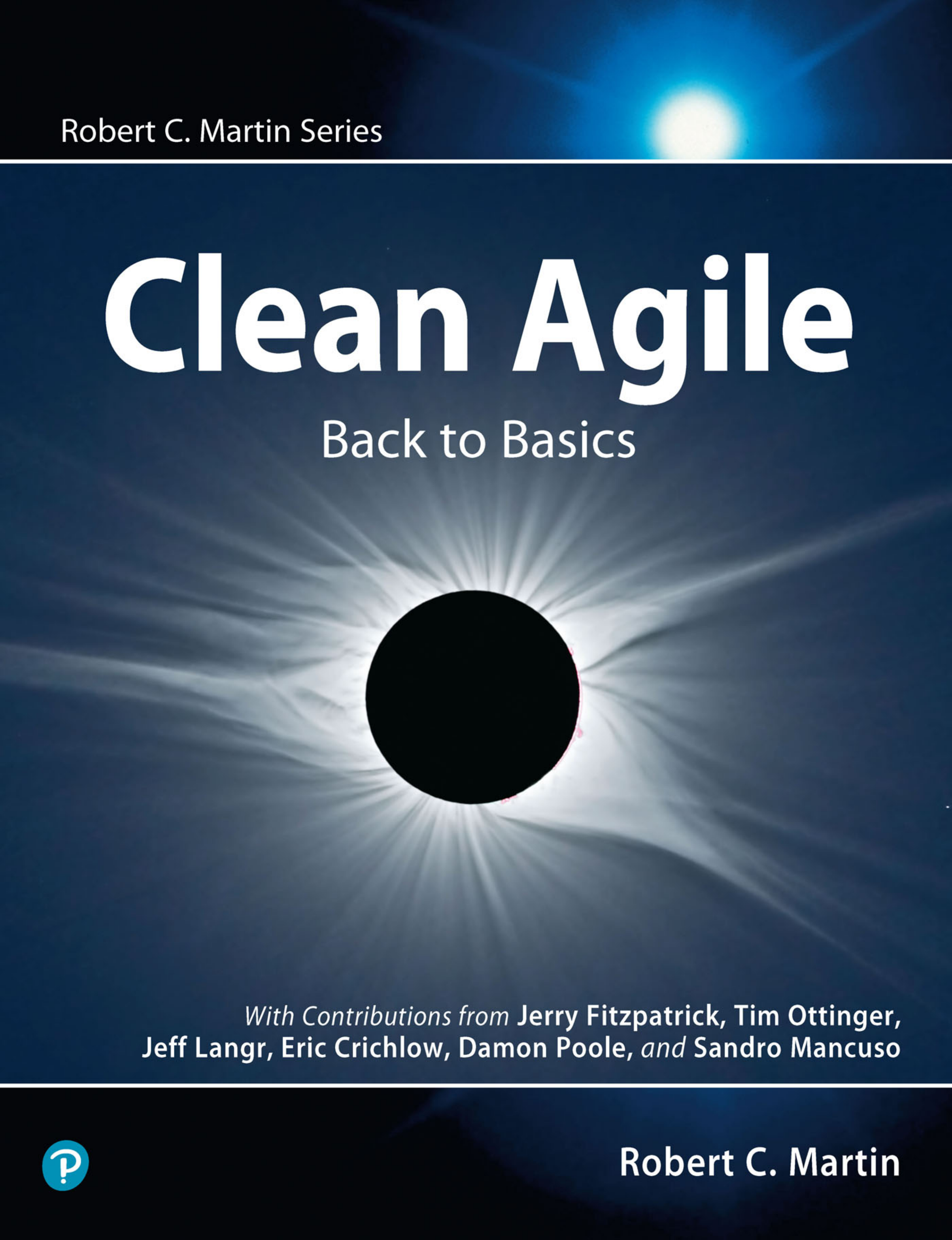

Robert C. Martin Series

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

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.