

DPI

*De* Programmatica *Ipsium*

DE PROGRAMMATICA IPSUM

---

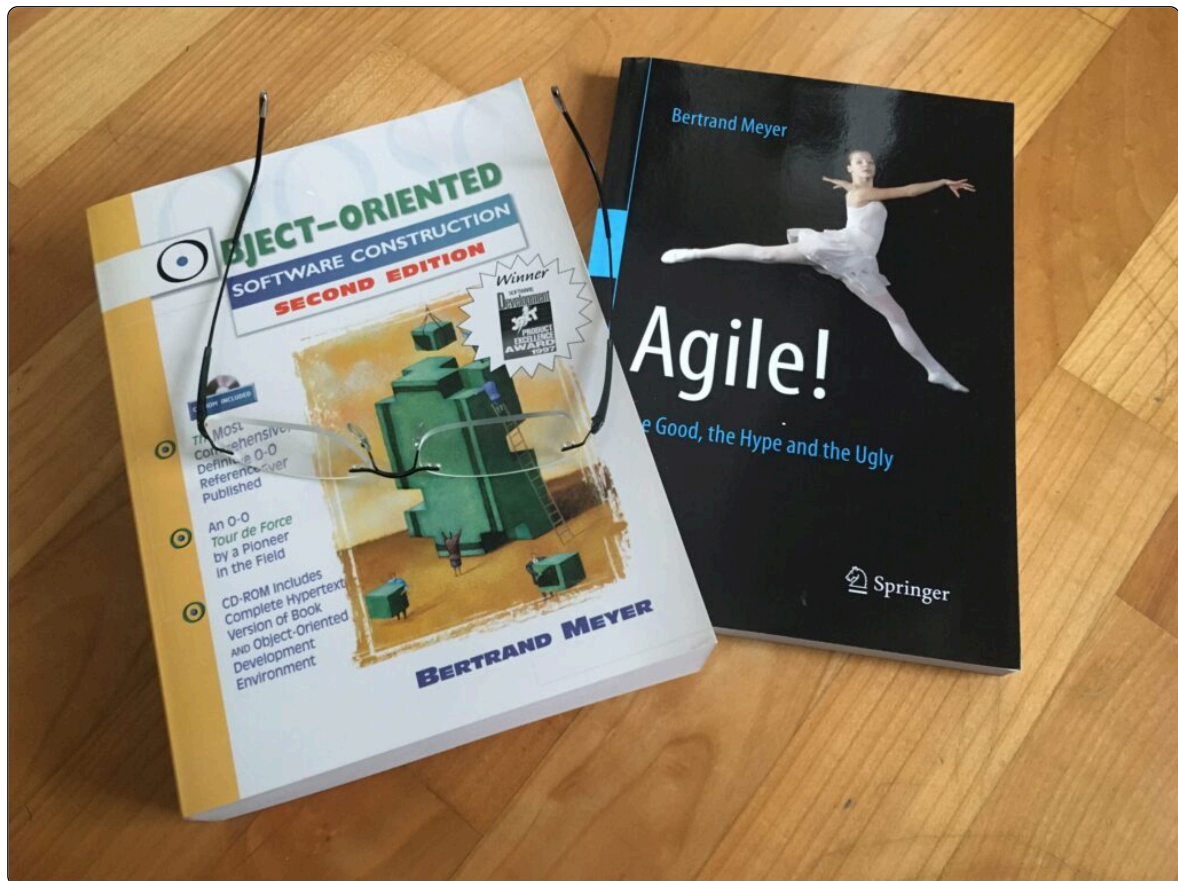
# Best Practices

# Table of Contents

Bertrand Meyer .....	5
Kent Beck .....	9
Steve McConnell .....	13
Douglas Crockford .....	21
Microsoft's Writings On Security .....	25
Robert L. Glass .....	33
Jenifer Tidwell .....	37
Michael Feathers .....	41
The Gang Of Four .....	45
Andy Clarke .....	51
Travis Swicegood .....	57
David Kadavy .....	63
Amy Brown & Greg Wilson .....	69
Scott Meyers .....	73
Josh Lockhart & Phil Sturgeon .....	81
Russ Olsen .....	87



# Bertrand Meyer



By Adrian Kosmaczewski, January 6th, 2020

When the author of these words started his career as a software developer, “object orientation” was all the rage. “Serious” programming languages were object oriented. “Professional” programming environments allowed one to view “objects” and “classes” in all of their glory. Inheritance, not composition, was the way of the future. Design patterns names were the answer to actual interview questions.

As many other developers, the path of this one started, however, with a humble programming language, sadly not yet dead<sup>1</sup>, called VBScript. In its version 5, released in 2000, this little mutant language featured a somewhat new concept

called “classes,” which at that time were more akin to C structs than anything else. These “classes” could not be inherited; well, to be fair they could not even have methods. They consisted of only data fields, and were more or less useful as a way to pass around large amounts of data back and forth from a Sub to a Function. In later versions (5.8 being the last one, apparently) this feature grew stronger and more complete<sup>2</sup>.

At that time, however, more interesting things were happening, like Java and its (at the time) clone called C#. Because, honestly, reading Java code starting with `public static void main(String args[])` was not much more difficult than reading `public static void Main(string[] args)` in C#. Thankfully things got more interesting starting with C# 2.0, released in 2005. But let us not digress.

The priority in 1999 was, then, to learn object orientation (and XML, as well.)

As fate wanted things to be, perusing the shelves of a bookstore in Buenos Aires, this author ended up buying a copy of Grady Booch’s Object-Oriented Analysis and Design with Applications, 2nd Edition<sup>3</sup> (OOADA.) Not a bad choice, actually; after all Booch is one of the authors of UML, so he knows a thing or two about objects. The best part of OOADA is the introduction, drawing ideas from psychology and sociology to explain what classes and objects meant. The final parts, featuring C++ code, are more arcane and difficult to follow for developers new to the language and the concepts (although a more recent 3rd Edition is said to feature Java code instead of C++.)

With hindsight, a much better choice would have been to acquire Bertrand Meyer’s<sup>4</sup> Object-Oriented Software Construction, 2nd Edition<sup>5</sup> (OOSC) instead. Not only because of breadth and depth (as suggested by a quick comparison of the page count, from 500 to 1200) and by language choice (Eiffel beating C++ in readability, particularly for a beginner in the field) but also, and most importantly, because of its *style*.

Bertrand Meyer is, hands down, the best writer in the computer field, because of a simple reason: his books have both great content *and* great prose. Not all authors of computer books (and certainly not the one you are reading now) can make the same claim.

OOSC is now considered a historic milestone in the history of object orientation, a major reference on the subject, all while remaining a delightful book to read. In this age where every language incorporates lambdas for the sake of functional programming fashion, this book remains a refreshing experience.

It provides excellent background information in various technical subjects, such as memory management, garbage collection, multiple inheritance, generic programming, type covariance, exception handling, and many other advanced topics. It also includes sections about analysis and style, and even a chapter on “Using inheritance well” and another on object persistence and databases. Needless to say, something many of us would need to read and re-read every so often.

Years later, as the hype of Agile programming methods was getting diluted in the rising reign of DevOps, Monsieur Meyer (still a teacher at ETHZ<sup>6</sup>) published a second gem: Agile! The Good, the Hype and the Ugly<sup>7</sup> (gotta love the exclamation mark, by the way.) This is probably the first serious book ever written about Agile (the second one being the long awaited and recently published Steve McConnell’s<sup>8</sup> More Effective Agile<sup>9</sup>.)

Ironically enough (or not,) Bertrand Meyer is not a signatory of the Agile Manifesto, resulting in a book free of hype or marketing propaganda. It consists of a short (170 pages) enumeration of “pros and cons” of various methodologies, with discussions about their applicability and suitability for certain projects.

In a time where Scrum seems to have become a *de facto* synonym of Agile, here is a book that actually sheds some light to the subject – and is actually a joy to read (By the way, if you are not interested in reading this book, at least watch this ACM Webinar<sup>10</sup> by Bertrand Meyer himself. You will not be disappointed.)

There is humor in the pages of these two books; there is wit, there is irony. But there is also depth, substance, and reflection. Merci beaucoup, monsieur Meyer.

Cover photo by the author.

### REFERENCES

<sup>1</sup> <https://isvbscriptdead.com/>

<sup>2</sup> <https://docs.microsoft.com/en-us/previous-versions//4ah5852c%28v%3dvs.85%29>

<sup>3</sup> <https://www.pearson.com/us/higher-education/product/Booch-Object-Oriented-Analysis-and-Design-with-Applications-2nd-Edition/9780805353402.html>

<sup>4</sup> <https://bertrandmeyer.com/>

<sup>5</sup> [https://en.wikipedia.org/wiki/Object-Oriented\\_Software\\_Construction](https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction)

<sup>6</sup> <https://ethz.ch/en.html>

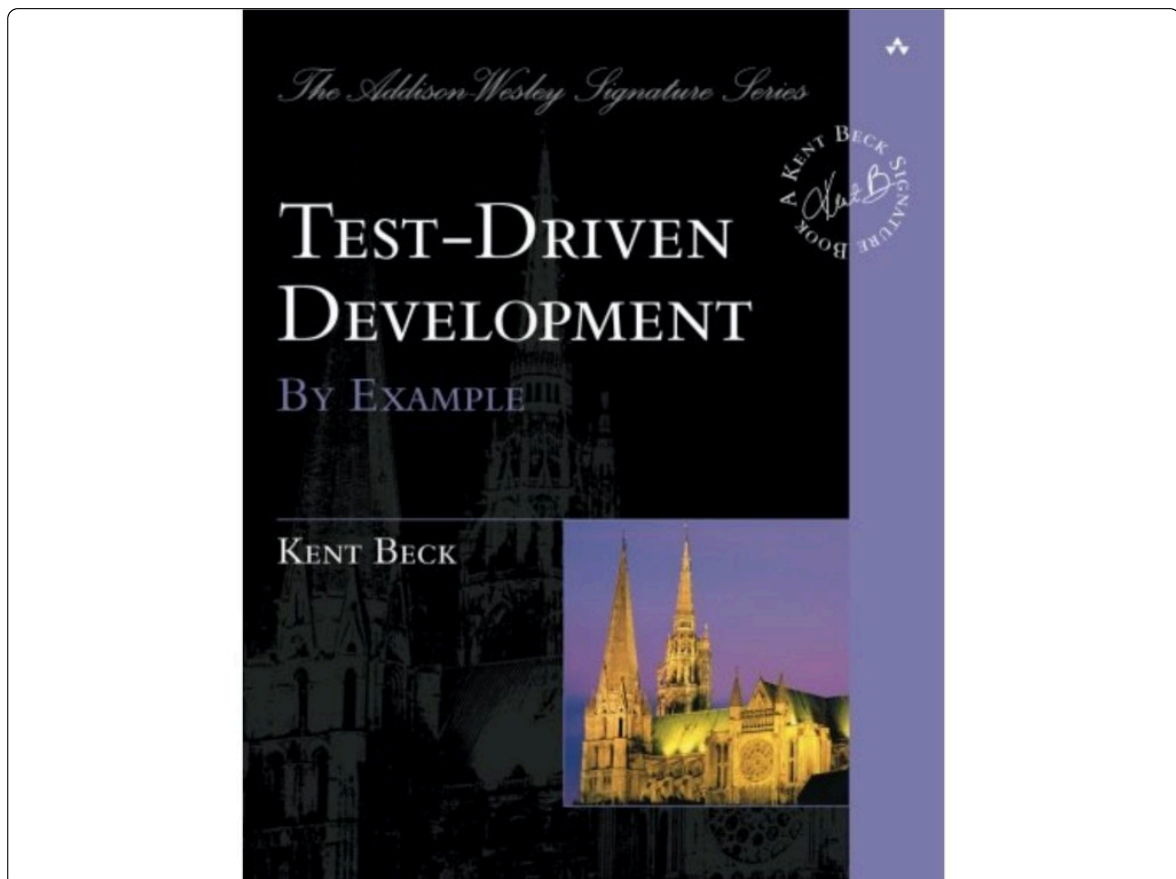
<sup>7</sup> <https://www.springer.com/gp/book/9783319051543>

<sup>8</sup> <https://stevemccconnell.com/>

<sup>9</sup> <https://moreeffectiveagile.com/>

<sup>10</sup> <https://learning.acm.org/techtalks/agile>

# Kent Beck



By Graham Lee, October 5th, 2020

Kent Beck might deny that Kent Beck needs an entry in the programmers' library. "All I did was rediscover what other people had done before," he might say, or "all I did was to interpret what Ward Cunningham was doing." But that discovery, that reinterpretation, is the most important part of the process. One person doing things differently is an oddball. Two are the beginning of a revolution.

This being the Smalltalk issue, I will start with one of Kent's books that deserves a wider audience: the freely-available "Smalltalk Best Practice Patterns<sup>1</sup>" (STBPP). Kent explained that when coding Smalltalk, he built a collection of index cards that

described design changes he could make to his code. If he was unhappy, he would riffle through the cards until he found a technique that might improve his work. The book represents a description of these techniques.

The word “patterns” appears in the book’s title as an example of what pattern languages in programming had been before everybody focused on the narrow collection of implementation patterns in the Gang of Four book. Any time there is something you need to do multiple times, and need to communicate to someone else that you are doing it or think they should do it, name it and describe it as a pattern. You are not limited to Abstract Factories and Singletons: Guard Clause is a pattern, and so is Intention Revealing Selector.

STBPP would have reached a broader readership and had more influence under the title “Pragmatic Refactoring”, but neither the name refactoring nor the “pragmatic” series had yet been created. It is not so much a book about things to do in Smalltalk, as a book about things to do when designing a program by building it.

Of course “designing a program by building it” also did not yet have a name when STBPP was written in the 1990s. That would come at the end of the decade, when Extreme Programming<sup>2</sup> was explained in “Extreme Programming Explained<sup>3</sup>”, by Kent Beck. XP would be one of the key influences behind the Manifesto for Agile Software Development<sup>4</sup>, a highly-influential document that counts Kent Beck among its co-authors and signatories (though he claims to have been ill and spent much of the time in his hotel room).

What have I missed out? Test-Driven Development<sup>5</sup>. CRC Cards<sup>6</sup>. Test && Commit || Revert<sup>7</sup>. Explore, Extract, Expand<sup>8</sup>.

Thanks, Kent.

## REFERENCES

<sup>1</sup> <http://stephane.ducasse.free.fr/FreeBooks/BestSmalltalkPractices/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf>

<sup>2</sup> <https://www.agilealliance.org/glossary/xp>

<sup>3</sup> <https://www.pearson.com/store/p/extreme-programming-explained-embrace-change/P100000657370/9780321278654>

<sup>4</sup> <https://agilemanifesto.org>

<sup>5</sup> <https://www.oreilly.com/library/view/test-driven-development/0321146530/>

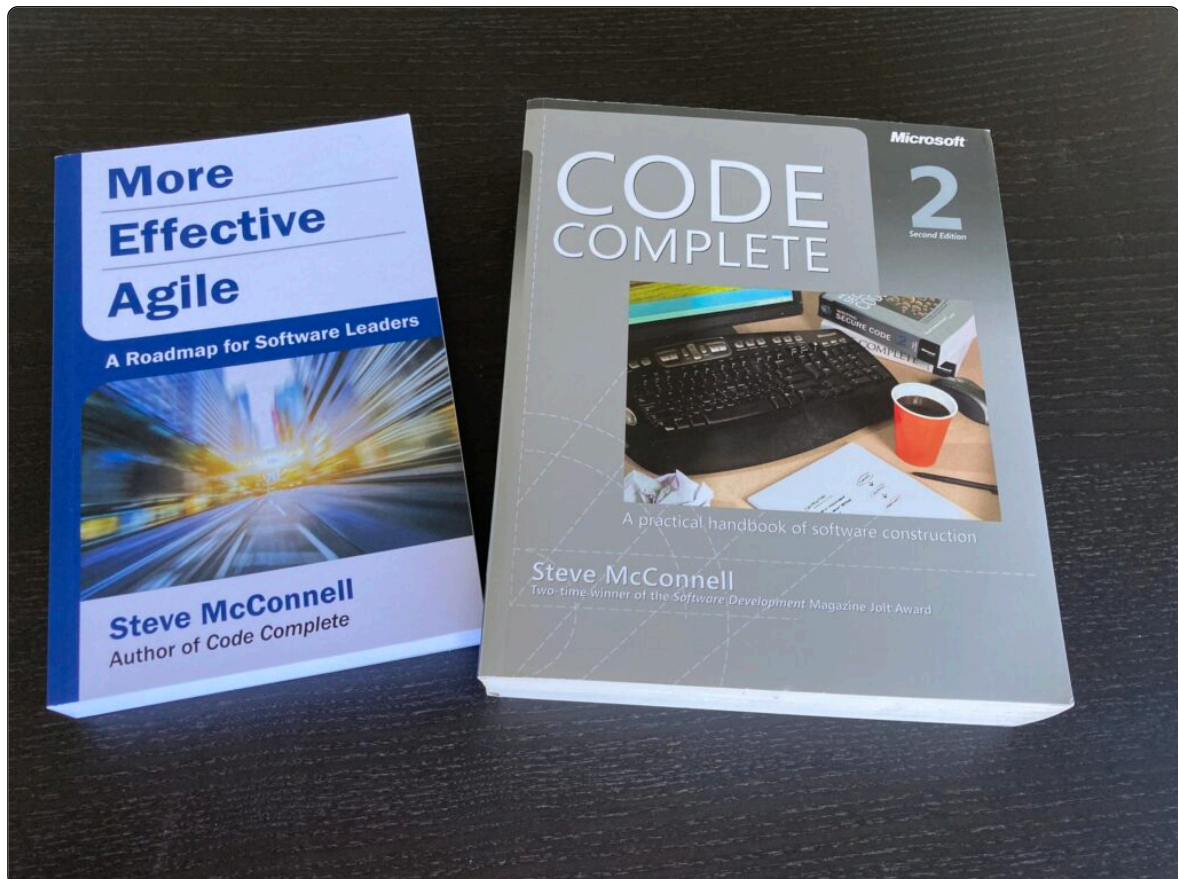
<sup>6</sup> <http://c2.com/doc/oopsla89/paper.html>

<sup>7</sup> [https://medium.com/@kentbeck\\_7670/test-commit-revert-870bbd756864](https://medium.com/@kentbeck_7670/test-commit-revert-870bbd756864)

<sup>8</sup> [https://medium.com/@kentbeck\\_7670/fast-slow-in-3x-explore-expand-extract-6d4c94a7539](https://medium.com/@kentbeck_7670/fast-slow-in-3x-explore-expand-extract-6d4c94a7539)



# Steve McConnell



By Graham Lee, December 7th, 2020

I almost wrote this article not about McConnell, but Microsoft Press. Why? Because developers always have something to learn, books have been a great way to share information for centuries, so reading about computing is central to the software engineering experience. If you do not believe me, reflect on the activity you are undertaking *right now*, reading an online magazine about computing.

You can tell the seriousness with which a platform company treats its developers—or its “developers, developers, developers”—by the tools, knowledge and support it gives them. Where some companies release their API documentation in bound

form through a recognised publisher, Microsoft just went and started their own book imprint. Not to bind their header comments between two boards, but to support professional programmers whatever their stripe (though maybe bear in mind which company has your back when choosing your technology, please).

Am I over-selling the importance of Microsoft Press, though? Steve McConnell, in *Code Complete, 2nd Edition* (2004, Microsoft Press), cites DeMarco and Lister's *Peopleware* when he tells readers "one book is more than most programmers read each year." He recommends trying to get through one every two months, as well as journals like *IEEE Software*, *Communications of the ACM*, and the now defunct and much-missed *Dr. Dobbs's Journal*. The "software developer's reading plan" at the back of the book contains 5 introductory-level books, 8 intermediate-level, and 7 leadership-level. It would take the median programmer (at least in 1999 when *Peopleware* was written) over 20 years to follow the plan, but a little over three years at McConnell's suggested rate. As long as you read *CC2E* first to know how quickly you are supposed to be reading!

Coincidentally, I left university to look for a job (which I found in the very room where I conducted my job search) in 2004, the year *CC2E* was published. Within a couple of years, when it was clear that my ability to solve problems using a computer was being hampered by my home-grown and BASIC-honed techniques at expressing computer programs, a manager gave me a copy of *Code Complete*. And the rest, as they say, is history.

There is lots to critique about the book, of course, particularly in 2020. Writing at the start of the Agile revolution, McConnell does not

have much to say about agile practices or methodology (two entries in the index, in fact, both pointing to single-paragraph descriptions of books; there is no discussion of agile in the main text). Sure, all of the parts are there—there is a distinct shift in the discussion of change management and risk between the first edition and the second, for example—McConnell acknowledges that the different activities in software construction occur in parallel, and that the earlier a change is handled, the cheaper it will be. But he never quite gets to the conclusion that software need not be "measure twice, cut once"; that unlike carpentry, software has a limitless supply of infinitely malleable wood.

In some of the places where McConnell uses multiple citations to make a claim about software engineering, other writers have found the citations based on questionable data or not generalisable to the way most software is written, particularly “classic” studies from the 1970s looking at programming batch systems. But we will save that story, because a future instalment of the Programmer’s Library will cover one of those books in detail. It is more interesting than it sounds!

There is also lots to recommend the book: in fact, in the 16 years since it was written, nothing else comes close. I still use and recommend practices described in *Code Complete*. I have internalised most of the ideas about code organisation and conception. When I see someone who reminds me of mid-aughts me, this is one of the books that springs to mind.

Other books have come and gone that try to do bits of what McConnell does here, with varying degrees of success. Robert Martin’s *Clean Code* on low-level design teaches similar ideas but with less intellectual rigour. The same goes for his *Clean Coder* on the practice of being a developer, though this is a more contested field with good showings from Pete McBreen (*Software Craftsmanship: The New Imperative*) and Hunt & Thomas (*The Pragmatic Programmer*). But none of these come close to supplanting *CC2E* as a manual for turning a programmer into a software engineer, and the realities of the software publishing business in 2020 mean that no replacement is likely to show up soon.

If all Steve McConnell had done was write this one book, his would be a significant contribution to the field. Even among the MS press books on software requirements, object thinking (not OOP), secure software development, solutions architecture, and more, this is a stand-out work. But he was and remains a prolific author, so now we turn to the rest of his library.

We will start by going back from 2004 to 1996. After the first edition of *Code Complete*, McConnell wrote a book about improving the capability of software teams to successfully deliver working software, ways of so doing he had uncovered by doing it and by helping others to do it. This book was *Rapid Development*, as distinct from the then-popular fad of Rapid Application Development. *Rapid Development* was explicitly *not* “programming, motherfucker”: an exhortation to just get down to coding as quickly as possible. *Rapid Development* was the (then,

and from what I have seen, now too) groundbreaking advice that you should maybe align your development practices with your customers' expectations and needs, and that doing so will help you give them what they need faster. This book had many recommendations for developers and managers to adopt on their team, including early and continuous delivery of working software through iterative, evolutionary development; continuous collaboration between the software team and the customers throughout the project; frequent feedback on the status and risks and reprioritisation so that the biggest risks are always next to be addressed; and more.

If this all sounds like it actually *is* agile software development, expressed four years before the Snowbird ski trip, that is for multiple reasons. Firstly by the time the manifesto was written, there was already broad understanding (at least among the more experienced and expert practitioners of software delivery) that the ideas of iterative, incremental development were transformative to the capabilities of software teams, there just was not yet a common banner under which all of the people calling for this could march. Secondly it is because I have been selective in my presentation of what is in *Rapid Development*: one of the “case studies” features a team who fail because they integrate their work too early, before all of the modules are fully tested and debugged, and this causes schedule slippage. Nowadays we recognise integration as one of the risks that we *should* mitigate by early and continuous monitoring; thus continuous integration.

But mostly it is because agile software development has been hollowed out since its introduction at the beginning of the millennium. Ask many development teams what they understand by Agile in 2020 and you will get a list similar to the highlights of *Rapid Development* I gave above: incremental and iterative delivery, customer involvement on the team, frequent retrospectives and feedback. That is not the entirety of the story, that is the Project Managers' Declaration of Interdependence<sup>1</sup> (originally published on Alistair Cockburn's site in 2005). Agile software development also included a number of *technical* practices, not explicitly called out in the manifesto, on the basis that what we do changes more frequently than why we do it.

We have already mentioned Continuous Integration, and developers might readily mention Test-Driven Development (and maybe its more agile siblings, Behaviour-Driven Development and Acceptance Test-Driven Development, which remind us that software is done not when the tests pass, but when the software does what the customer wants from it). Some may begrudgingly allow daily stand-up meetings as an agile practice, though they may grumble. But how about user stories? Those are a technical practice, and an example of specification-by-customer-conversation. Domain-Driven Design encourages a shared vocabulary between the software team and the customers, enabling those conversations. A review of agile practices in scientific software development<sup>2</sup> lists 35 practices explicitly mentioned in the Scrum and XP methodologies. Of those, I informally count 22 technical (or at least not purely project-management) practices, and I would also suggest that those technical practices are the ones most patchily adopted across the industry, based on my own experience in many teams that describe their work as “Agile”.

*Rapid Development* gets a bye on some of those practices on the basis that they had not been invented in 1996, but also because McConnell makes a much more general and much more useful argument: technical practices *are not guaranteed* to improve your ability to deliver working software to your customers. Indeed adopting new practices on a new project is likely to *reduce* your control over and understanding of the project, thus *harming* your ability to communicate effectively with customers what you are capable of, how much it will cost, and how long it will take. This incredibly simple and important notion has been rediscovered many times in the industry, and is currently usually cited in the form of “innovation tokens” from Dan McKinley’s choose boring technology<sup>3</sup>.

*CC2E* gets a bye on its short shrift on Agile for the innovation tokens reason: before the agile-industrial complex took over and made agile all about hiring scrum masters and producing burn-down charts, it was a shorthand for a lot of project management *and* technical practices and there was not yet compelling evidence that taken together, those practices improved the life or capability of a software engineer.

Maybe there is not, but we can turn back to Steve McConnell for a modern discussion with his 2019 book, *More Effective Agile*<sup>4</sup>. *More Effective Agile* is the

*Rapid Development 2.0* for the agile-industrial complex, though we should note that the world is in a much better place when it comes to software delivery than it was in

1996. Now the question most people have is not “why do my software projects always fail“, but “why do my software projects always cost a bit more, or take a bit longer, than I would like”. Eventually many people end up with the question “why has all my forward progress ground to a halt in a mire of technical debt and defect fixing”, but often answers to those questions can be deferred for the successor CTO to answer.

In this context, McConnell revisits the central premise of *Rapid Development*: what does a software team need to succeed, are they getting it, and how can they get it? He describes Scrum as a good baseline framework for organisations transitioning to the agile approach, and “scrumbut” as a common reason for novice adopters to fail. Scrumbut is a hybrid methodology, where you treat Scrum as an a la carte selection of practices to pick and choose as you see fit. “We do scrum, but we have our daily standups once a fortnight”. “We do scrum, but our scrum master is also our engineering manager”.

Does this make McConnell a member of the brinksmanship school of the agile-industrial complex: if agile is not working for you, it is because you are not agiling hard enough? No, it makes him a realist: scrum is a framework for process improvement, but you can only succeed at improving your process once you have measured what is wrong with it and hypothesised how to fix it. If you are just starting out, stick in the shallow end of the pool and adopt the baseline process. One that is working for you and you are not going to drown, it is time to try changing things.

Just as agile itself jettisoned many of the agile practices from methodologies like XP, and *Rapid Development* borrowed from Fred Brooks the idea that there is “no silver bullet”, so *More Effective Agile* does not mandate specific technical practices. The two that are given decent space in the book are Continuous Integration and Continuous Delivery, both of which are still unevenly distributed throughout the industry so are good choices for an author who wants to make agile teams more effective.

Steve McConnell has been working in software and teaching software practitioners for over three decades, and there is still much to learn.

Cover photo by Adrian Kosmaczewski.

REFERENCES

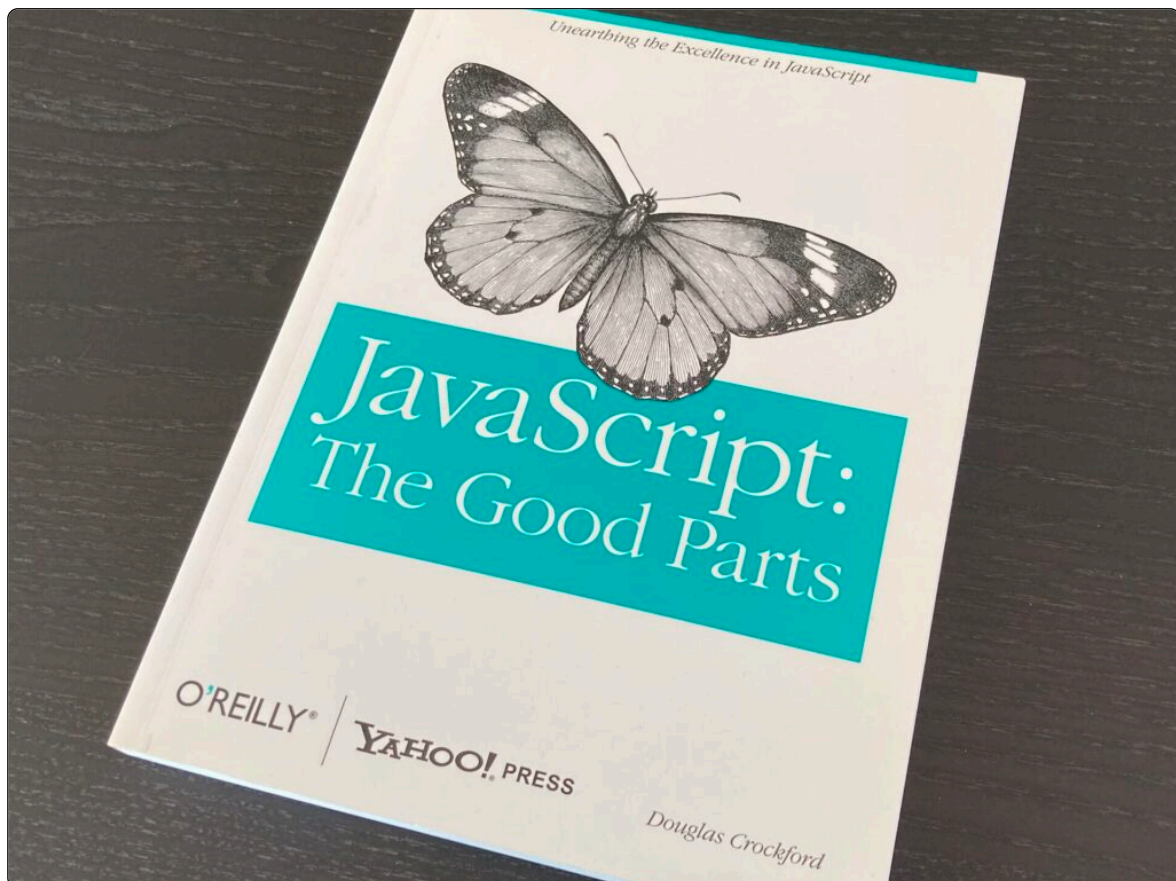
<sup>1</sup> <https://www.adventureswithagile.com/2014/08/19/declaration-of-interdependence/>

<sup>2</sup> <https://dl.acm.org/doi/10.1145/1985782.1985784>

<sup>3</sup> <https://mcfunley.com/choose-boring-technology>

<sup>4</sup> <https://moreeffectiveagile.com/>

# Douglas Crockford



By Adrian Kosmaczewski, September 6th, 2021

O'Reilly published in 2020 the seventh edition of one of the biggest bestseller programming books of the past 25 years, Flanagan's "JavaScript: The Definitive Guide"<sup>1</sup>. At 700 pages and weighing 1.2 kg, it is a book that easily stands out in any good programmer's library. Many developers have used such information to joke about the fact that the good parts of JavaScript, as catalogued by Crockford in his eponymous 2008 book<sup>2</sup>, is merely 180 pages long, and weights only 290 grams; that is, only 25% of JavaScript is actually any good.

That comparison is misleading, to say the least, particularly because Crockford's book also includes a colophon, a dedication, a preface, lots of syntax diagrams, and two appendices called "Bad Parts" and "Awful Parts". All in all, the good parts are just 140 pages, give or take.

Before the book, Crockford was already a celebrity for many of us dealing with JavaScript in a daily basis. This author came across his humble web page<sup>3</sup> around 2002, still available at the time of this writing, where he claimed that JavaScript was "The World's Most Misunderstood Programming Language". The revelation was not small, and in hindsight, was a major event for the software industry.

The issues with JavaScript, of course, start with the name. The language that Brendan Eich baptised LiveScript was renamed to JavaScript during the dotcom craze of the 1990s, because marketing and NASDAQ and Java and IPOs and Sun and Netscape and whatnot. These days LiveScript is an actual different thing<sup>4</sup>, but still related to JavaScript, just to make things more confusing for everyone. But the industry remained confused about the subject of Java vs. JavaScript for decades. Jeremy Keith summarized the situation<sup>5</sup> in one phrase: "Java is to JavaScript as ham is to hamster."

Then there were paradigm issues; Java was the apotheosis of object orientation. JavaScript... has a new keyword, but one which does not work exactly the way a Java developer would expect. And it has a weird<sup>6</sup> inheritance model. And of course object orientation in the 1990s was all about inheritance and nothing about composition, because few had read Barbara Liskov's paper<sup>7</sup> yet. So if JavaScript's inheritance was not in the flavor of Java's or C++'s, clearly it was inferior. To illustrate the confusion, Crockford even explained how to encapsulate private fields<sup>8</sup> with JavaScript in his website.

Finally, there were quirks. There were many<sup>9</sup>. JavaScript could use semicolons, or not; there were `String` and `string`, and nobody quite understood the relationship between both; there was `==` and `===` and nobody quite understood the difference between both. And the list could go on and on and on.

(These were the subjects developers argued about in Slashdot<sup>10</sup> or Evolt<sup>11</sup> back in those days, long before Reddit or A List Apart were a thing.)

Crockford's website<sup>12</sup> (and later, book) was arguably one of the igniters of the renaissance of JavaScript. In the wake of the dotcom bubble crash<sup>13</sup>, as web developers were struggling to find a job, Crockford realized that what we needed was to calm down for a minute, have a cup of tea, and realize that JavaScript was *different*. Just that; not better, not worse, just different, and that no, it did not suck<sup>14</sup>. We just needed to take a breath, and stop screaming the words "startup" and "IPO" to every venture capitalist walking in the street.

If this was not enough for his résumé, let us not forget that Crockford also introduced JSON<sup>15</sup> and JSLint<sup>16</sup> to the world.

But we can go even further, without exaggeration or hyperbole: in retrospective, the biggest contribution of Crockford was to finally make functional programming mainstream. It was this book, together with the rise of Ruby on Rails, what would make millions of developers all over the world, including the author of these lines, understand what a closure was for the first time. And since, according to Herb Sutter at least, the free lunch was over<sup>17</sup>, the time was ripe for functional programming to rise.

Cover photo by the author.

### REFERENCES

- <sup>1</sup> <https://www.oreilly.com/library/view/javascript-the-definitive/9781491952016/>
- <sup>2</sup> <https://www.oreilly.com/library/view/javascript-the-good/9780596517748/>
- <sup>3</sup> <https://www.crockford.com/javascript/javascript.html>
- <sup>4</sup> <https://livescript.net/>
- <sup>5</sup> <https://www.smashingmagazine.com/2009/07/misunderstanding-markup-xhtml-2-comic-strip/>
- <sup>6</sup> <https://www.crockford.com/javascript/prototypal.html>
- <sup>7</sup> <https://deprogrammaticaipsum.com/barbara-liskov/>
- <sup>8</sup> <http://www.crockford.com/javascript/private.html>
- <sup>9</sup> <https://quirksmode.org/>
- <sup>10</sup> <https://slashdot.org/>
- <sup>11</sup> <https://evolt.org/>
- <sup>12</sup> <https://www.crockford.com/javascript/>
- <sup>13</sup> [https://en.wikipedia.org/wiki/Dot-com\\_bubble](https://en.wikipedia.org/wiki/Dot-com_bubble)
- <sup>14</sup> <https://www.sitepoint.com/interview-doug-crockford/>
- <sup>15</sup> <https://www.json.org/json-en.html>
- <sup>16</sup> <https://en.wikipedia.org/wiki/JSLint>
- <sup>17</sup> <http://www.gotw.ca/publications/concurrency-ddj.htm>

# Microsoft's Writings On Security



By Graham Lee, October 4th, 2021

Yes, you read that correctly. Microsoft. Writing on information security. They may be the software company who have done the most writing on information security, including many security software companies.

The reason they did it is one of those things that made Bill Gates such a capable and successful business leader. He was not necessarily a product visionary, though he

did come up with the “a computer on every desktop” vision that made Windows and Office ubiquitous. No, Windows and Office were not the first examples of products in their field to market. What Bill Gates was great at was seeing what he and his team were not great at, and doing that.

In 2002, this was security. Various worms and viruses plagued the Windows ecosystem. These had names that we recognise now, like Nimda and Code Red, which helped to cement their place in the public consciousness in those as Spectre and Meltdown have elevated the publicity of some CPU design flaws today. The “ILOVEYOU” virus even got debated in the House of Commons<sup>1</sup> of the UK parliament, as a national defence issue.

Microsoft could probably have done nothing, and been OK for a while. Through a combination of an improved product and shady practices—pricing OEM licenses unfavourably if the vendors offered alternative operating systems—Windows 95 had already won an overwhelming share of the desktop computer market, a position which Windows XP had solidified.

When you run the whole market it can be difficult to see a need to change direction, but Bill Gates could do it. He did it in 1995 with his Internet Tidal Wave<sup>2</sup> memo: Microsoft were late to the party, but they were going to show up in a massive stretch Hummer and spike the punch.

He did it again in 2002 with his Trustworthy Computing<sup>3</sup> memo.

*Over the last year it has become clear that ensuring .NET is a platform for Trustworthy Computing is more important than any other part of our work. If we do not do this, people simply will not be willing – or able – to take advantage of all the other great work we do. Trustworthy Computing is the highest priority for all the work we are doing. We must lead the industry to a whole new level of Trustworthiness in computing.*

Software engineers across the whole of Microsoft downed tools (there was no new development on Windows for two months), learned how to build things in a security-conscious fashion, then invested time into building threat models, identifying risks, and patching their software. Never before or since has a software company

so totally—and publicly—admitted its faults, and come to a halt while it addresses them. And along the way, we got some useful information about how to do the same ourselves.

*We've also published books like "Writing Secure Code," by Michael Howard and David LeBlanc, which gives all developers the tools they need to build secure software from the ground up.*

So BillG even has some recommended reading for us. Shall we take a look?

Recommended for us but required reading for Microsoft engineers, *Writing Secure Code* explains why secure systems are needed, gives a light touch description of threat modeling, then goes into a load (440 pages) of secure coding techniques. Many of these techniques are still needed today, because they are design-level (access control, execution privilege, appropriate application of cryptography) or because we never learn (buffer overruns, data representation, code injection). For example, their description of "Canonical Representation Issues" covers the various ways in which Unicode text can look like other Unicode text, a problem in 2002 that foreshadows the rise of punycode domain spoofing that still plagues the Firefox browser in 2021.

Why are these things, these important security aspects of application and system design, still a problem two decades later? Why is Anastasiia Vixentael, the guest author in our security issue<sup>4</sup>, still explaining to developers how to do (and that they should do) these things that were known back before many React Native coders were even conceived?

The answer is found in the excellent and evergreen Appendix B to Howard and LeBlanc's book, "Ridiculous Excuses We've Heard". This covers reasons Microsoft teams gave the authors why they should not need to adopt secure coding practices, along with counter-arguments to each of them. Yes, I have been the information security expert on many a team in the 21st century, and yes, I have heard many of these ridiculous excuses.

Of course the ridiculous excuses are there to cover the vulnerability of the developers, project managers, product owners, testers, and other engineers: they were never

taught about information security or secure development practices. Telling them it is the most important thing to the company is telling them their most pressing need is one they are ill-equipped to handle, and they get defensive. In the introduction to *Writing Secure Code*, Howard and LeBlanc make the simple observation that the most popular MS Press book on programming, *Code Complete 2*<sup>5</sup>, does not mention security once.

I will save you some trouble by modernising that statement: programmers still are not told a lot about security as part of software development. *Succeeding with Agile* does not tell you to adopt secure coding practices, or build security into your backlog. *Continuous Delivery* tells you that “capacity and security requirements” might exist multiple times, but its only specific recommendation is to use a linter to catch common problems. *The Pragmatic Programmer* does have a topic, “stay safe out there”, explaining basic security principles. The Pragmatic approach, apparently, is to let somebody else worry about it. Let us hope that some engineers never read that book, or we are a generation or two away from everybody hoping that somebody else is “doing security” for them.

Michael Howard also co-authored a book with Steve Lipner, also in the “Microsoft secure software DEVELOPMENT SERIES” as the cover styles it, called *SDL: The Security Development Lifecycle*. This book could be treated as a timeless guide to designing security into a software development process. There is a list of banned functions that you might want to update, and a Visual C++ solution on a companion CD that will only work in Visual Studio 2005 apparently, but other than that everything in the book is about how you incorporate security thinking into the other thinking you are doing.

Thinking does not change that much. We have already seen that specific code-level problems identified in *Writing Secure Code* are still problematic today. So, it turns out, are brain-level problems. Even at ISO 27k-certified companies (the standards series for information security management systems) you will find developers who think that because they are using JWTs, they are secure<sup>6</sup>; and product owners who see all that data coming in and ask who it can be sold to.

The authors of the SDL take us through everything we need to do to integrate security thinking into our software efforts. Starting with before the project kicks

off (education, awareness, leadership support), to evaluating whether an SDL is needed, designing security requirements, building and testing them, and making sure software is secure in deployment. They even tell you how to integrate the SDL into those new-fangled Agile™®© methods everyone was raving about in 2002. The description is golden:

*We've heard people claim that <insert popular development method> produces bug-free software. This might be true—and of course, it is true if you know nothing about security bugs, because you wouldn't recognise a security bug if you had no idea what one was.*

There is a bit missing from our story, though. We know how to plan the work into our project, and we know how to address code-level issues, but how do we know what the risks are? How do we understand what the security posture of our app is?

For that, we turn to the third and final of today's readings, *Threat Modeling* by Frank Swiderski and Window Snyder. Snyder<sup>7</sup> is a bit of a legend in software security circles. She was at Microsoft for the trustworthy computing initiative, and has held security leadership positions at Apple, Square, Fastly, Intel, and Mozilla (where her title was CSSO: “Chief Security Something-or-Other”). She is currently working on a startup, Thistle Technology, working to improve security of IoT<sup>8</sup>.

The benefit of the *Threat Modeling* approach is that it is easy to understand and apply. Swiderski and Snyder have handy mnemonics for remembering important ideas. Threats are classified based on their effect with the STRIDE acronym: Spoofing, Tampering, Repudiation, Denial of Service, or Elevation of Privilege. Vulnerabilities and risks are ranked based on their DREAD scores: Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability.

Elsewhere, the threat modeling process uses skills your team probably already has. How do you identify whether an attacker can leverage some asset using some particular entry point? Build a data flow diagram; your engineers probably already do this on a whiteboard whenever they need to discuss the behaviour of some API call. How do you know what is there to be attacked? Create use scenarios: you probably already have these, and they are probably written in the form “AS A journalist, I WANT TO...”.

Unfortunately Microsoft Press does not carry an up to date equivalent of any of these books. The second edition of *Writing Secure Code*, from 2003, is still the edition you can buy on their store. You will not find the other two at all; not even the SDL book which only needs someone to port the Visual Studio solution to a modern IDE.

We know that security problems have not gone away, so why are not Microsoft still providing leadership here? Maybe they think their online security development lifecycle guide<sup>9</sup> is sufficient, even though nearly half of the resources are in the Legacy Archive<sup>10</sup>. Maybe thinking about security was a fad, and the people involved are now all thinking about reasoning in type systems or shoveling even more un-audited javascript into everybody's computers at Microsoft subsidiary npm, Inc. Maybe the original Trustworthy Computing push achieved its goal, and moved the heat onto other vendors.

Whatever the case, software security has not gone away, and its discussion should not be limited to black hat nerds and CISSPs in suits opening `calc.exe` on each other's laptops in DefCon meetings. Do yourself and your customers a favour, and find copies of these excellent works in your favourite second hand book store.

Image by the author.

## REFERENCES

- <sup>1</sup> <https://hansard.parliament.uk/Commons/2000-07-04/debates/2aee8540-6e7f-4fdb-b620-0b8b6e410298/ComputerViruses?highlight=love#contribution-9d212745-cadb-44b3-a256-55ce707b7c09>
- <sup>2</sup> <https://www.wired.com/2010/05/0526bill-gates-internet-memo/>
- <sup>3</sup> <https://news.microsoft.com/2012/01/11/memo-from-bill-gates/>
- <sup>4</sup> <https://deprogrammaticaipsum.com/secure-development-is-dead-long-live-secure-development/>
- <sup>5</sup> <https://deprogrammaticaipsum.com/steve-mcconnell/>
- <sup>6</sup> <https://jwt.io>
- <sup>7</sup> [https://en.wikipedia.org/wiki/Window\\_Snyder](https://en.wikipedia.org/wiki/Window_Snyder)
- <sup>8</sup> <https://techcrunch.com/2021/04/22/thistle-technology-seed-security-iot/?guccounter=1>
- <sup>9</sup> <https://www.microsoft.com/en-us/securityengineering/sdl/>
- <sup>10</sup> <https://www.microsoft.com/en-us/securityengineering/sdl/resources>



# Robert L. Glass



By Adrian Kosmaczewski, November 1st, 2021

Robert L. Glass wrote a book in 1998 called “Software 2020”<sup>1</sup>, currently rated with only one star... by the author<sup>2</sup> himself. In his review<sup>3</sup> (written in 2017) he justifies this abysmal record because of a simple observation: none of the predictions in the book turned out to become a reality.

On the other hand, “Facts and Fallacies of Software Engineering”<sup>4</sup>, published in 2003, has four stars out of five on Amazon. This opus, then, clearly outperforms the previous title by the same author. Maybe this success is due to the fact that, unlike its predecessor, this book does not predict the future; instead, it describes

an eternally grim present, one that never seems to go away, no matter how many new versions of your debugger you try.

Because, let us be honest. Reading this book is pure masochism. Not because it is badly written (far from that) or that the contents are not relevant. Quite the opposite, actually; the sad truth of our craft is described in painful detail in every single section. Reading it is an exercise in nodding.

This book has received its fair share of reviews<sup>5</sup>, Slashdot threads<sup>6</sup>, and discussions<sup>7</sup>. No need to indulge the reader into yet another discussion of the relative merits of each section, or whether the author agrees with each item. That is utterly irrelevant.

The core question raised by this book, re-reading it in *Anno Domini* 2021, is why is our industry still falling into the same traps? Why do managers keep considering lines of code as a valuable metric (fallacy 6)? Why is it that companies need to continuously break<sup>8</sup> backwards compatibility in their SDKs and IDEs (fact 6)? Why are schools not properly training younger developers to the reality they will face during their careers (fact 48 and fallacy 10)? Why is it that full-stack developers feel an urge to dive every year into the newly hyped<sup>9</sup> JavaScript framework *du jour* (fact 5)? Why are not product owners aware of the increase in complexity of every new feature added to the backlog (facts 21 and 23)? Why do some companies insist in 100% test coverage, even though it is not useful nor enough (fact 33)? Why do we keep on insisting on cramming developers in open space offices (facts 4 and 13)? Why do many teams still not adopt healthy code review practices (fact 37)?

Why is it that our craft is such a disgusting mess of appalling whimsical cargo-cult practices, disguised as perfectly coherent and rational ideas? A deep sigh ensues.

Not being able to resist the temptation, this author will briefly mention one fact and one fallacy to start reading:

- Fact 30: COBOL is a very bad language, but all the others (for business data processing) are so much worse.
- Fallacy 8: “Given enough eyeballs, all bugs are shallow.”

This book can be read in a fully non-linear fashion, so feel free to explore this classic starting with these two. Think of this book as a guide, providing answers for most

questions (almost all, I would say) you are going to have about your craft, your teams, and your code, for years to come.

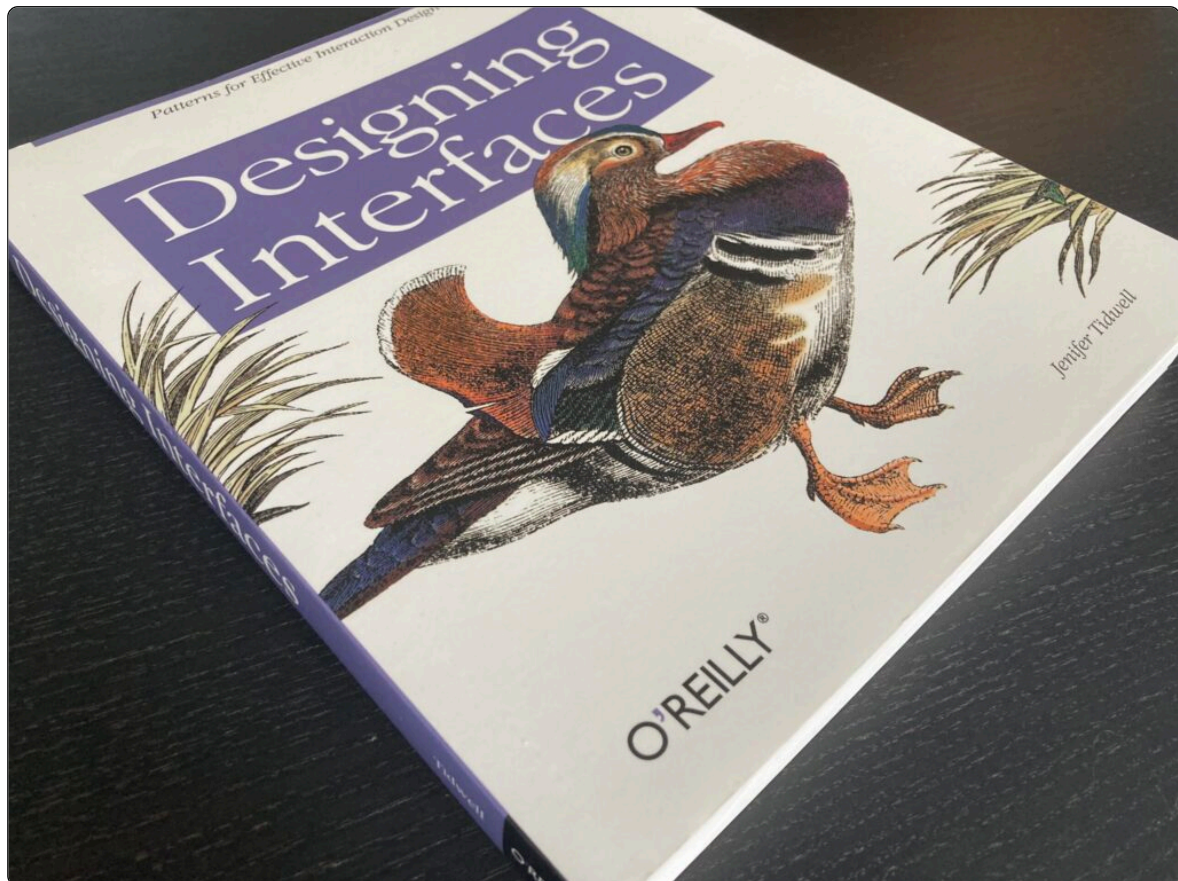
As a personal wish, the author of these lines can only hope for a future in which our craft will work on each of these facts and fallacies, and work towards fixing them in the long run.

Cover photo by the author.

### REFERENCES

- <sup>1</sup> <https://www.amazon.com/exec/obidos/ASIN/0942337050/developerdots-20>
- <sup>2</sup> [https://en.wikipedia.org/wiki/Robert\\_L.\\_Glass](https://en.wikipedia.org/wiki/Robert_L._Glass)
- <sup>3</sup> [https://www.amazon.com/gp/customer-reviews/R1TEBKSHE41A50/ref=cm\\_cr\\_dp\\_d\\_rvw\\_ttl?ie=UTF8&ASIN=0942337050](https://www.amazon.com/gp/customer-reviews/R1TEBKSHE41A50/ref=cm_cr_dp_d_rvw_ttl?ie=UTF8&ASIN=0942337050)
- <sup>4</sup> <https://www.oreilly.com/library/view/facts-and-fallacies/0321117425/>
- <sup>5</sup> <https://blog.codinghorror.com/the-delusion-of-reuse/>
- <sup>6</sup> <https://slashdot.org/story/04/08/27/1616256/facts-and-fallacies-of-software-engineering>
- <sup>7</sup> <https://stalk-calvin.github.io/blog/2017/03/25/software-facts-fallacies.html>
- <sup>8</sup> <https://deprogrammaticaipsum.com/issue-18-obsolence/>
- <sup>9</sup> <https://deprogrammaticaipsum.com/issue-1-hype/>

# Jenifer Tidwell



By Adrian Kosmaczewski, January 3rd, 2022

If there is one thing that computer books are most definitely not usually praised for, it is their visuals. Thankfully, books about user experience and user interface design are usually, indeed, worthy of such acclaim. In this case, however, limiting a review to such criteria would be short-sighted, poor, and unjust. The truth is that most important literature works are multi-layered, profound, and suitable for multiple relectures.

In 2007, when Jenifer Tidwell<sup>1</sup> published the first edition of her book “Designing Interfaces” (its third edition<sup>2</sup> was published January 2020), we had been, as an in-

dustry, making visual user interfaces on bitmap displays for around 45 years. Well, maybe not for *that* long. Yes, it all started with Ivan Sutherland's "Sketchpad"<sup>3</sup> PhD thesis at the MIT, but GUIs remained a niche curiosity for two decades, while Douglas Engelbart<sup>4</sup>, Alan Kay<sup>5</sup>, and Jef Raskin<sup>6</sup> worked on it. It was not until the Mac in the 1980s, and later Windows and the Web in the 1990s, that GUIs became what they are today.

So it is fairer to say that Ms Tidwell's book encapsulates roughly the first quarter century of GUI knowledge ever produced by the human race; quite an accomplishment in itself.

All of this knowledge is organized as a collection or catalog of "patterns"; by far one of the most hyped<sup>7</sup> words in programming literature between 1995 and 2007. Starting with the overall architecture of a user interface in panels and windows, Ms Tidwell drilled down to the most common elements, such as preview panes, button groups, autocompletion, breadcrumbs, treemaps, and more. All in all, almost 100 different patterns; 94 to be exact. Each profusely illustrated with examples from the Motif, Xerox Star, Mac ("Classic" and OS X), and Windows galaxies, but also showcasing PDA screens, and even examples of popular cellphones of the pre-iPhone era.

The first layer of contact with the book, the visual production, hits your brain as a high-speed train as soon as you have it in your hands; it is hard to browse this gem and not dream of designing the next laureate of the Apple Design Awards<sup>8</sup>. Few books in our craft produce such effect. It is refreshing.

On a second appreciation, comes to the reader the size and breadth of the catalog. Similarly to other books<sup>9</sup> built around the "pattern" moniker, Ms Tidwell crafted a reference book that has its place next to the keyboard and mouse of anyone creating software for a living or for pleasure. The logic behind each visual element is clearly explained, and documented with examples coming from the most radically different environments, some of which are of historical relevance at this point.

The third, and in the opinion of this author, the most important takeaway from the book is not the (gorgeous) visual design, nor the immense wisdom of the pattern catalog; it is rather in the following statement in the opening pages:

*The first step in designing an interface is figuring out what its users are really trying to accomplish.*

Ah, the most important question, the one that many still fail to answer before 1.0 hits the road.

2007 was a time of brushed metal<sup>10</sup> interfaces on Aqua<sup>11</sup>; of translucent Windows Vista title bars on Aero<sup>12</sup>; of Web 2.0 candy-like UIs with shadows and gradients; and of the CSS Zen Garden<sup>13</sup>. It was also the year of the unveiling of the iPhone, arguably<sup>14</sup> the most successful product of all time, and one that had quite an impact in the design of space-constrained user interfaces. The first edition of this book, of course, does not mention it at all, and understandably so.

Fifteen years later, we live in a world where Atwood's Law<sup>15</sup> became the norm, where interfaces are reactive (whatever that means), and where "flat" is a shiver-inducing visual trend. Here is this author hoping that the creators of the next Electron app will get a copy of Ms Tidwell's book, dive into it, and find out that there is indeed an answer for pretty much every single question they might have. User interfaces do not require any more shallow, mindless "innovation"<sup>16</sup> at this point.

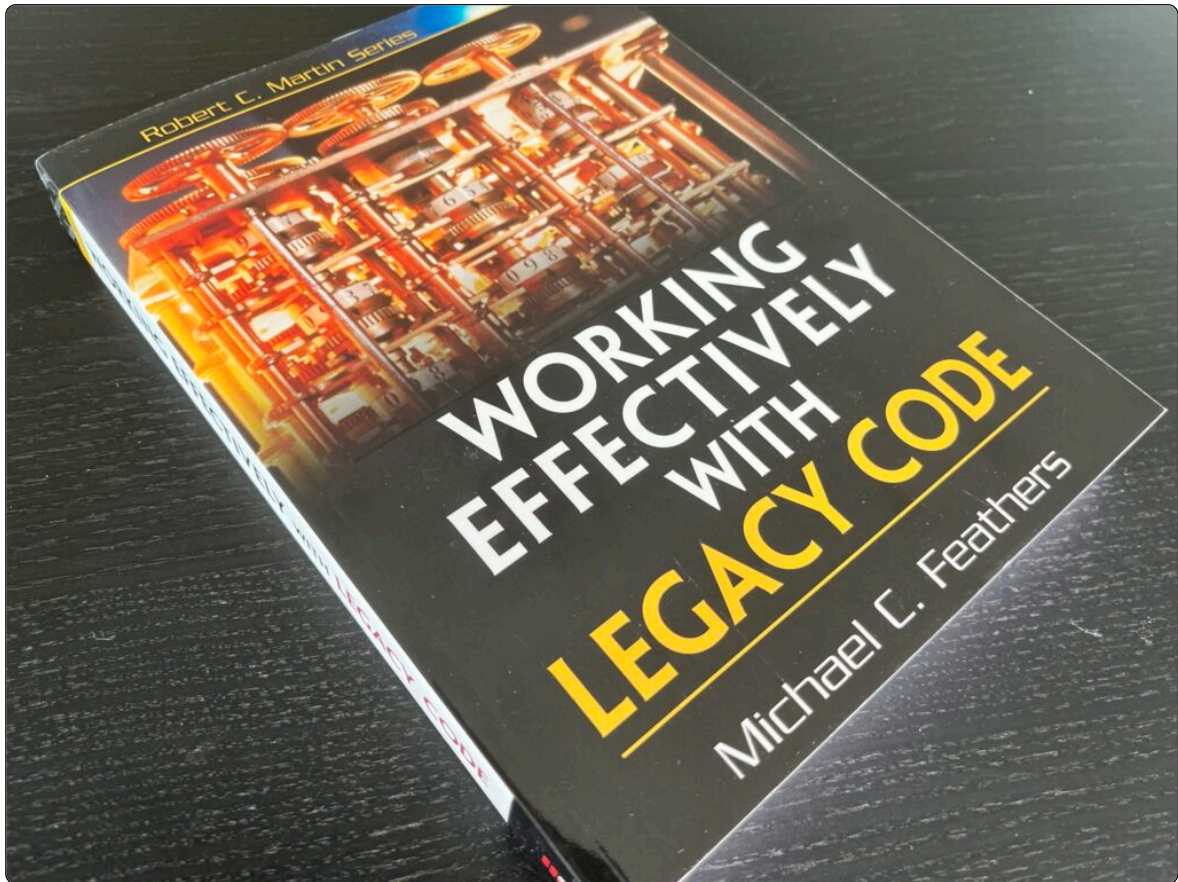
For those programmers interested in visual design, this author recommends coupling this book with Jeff Johnson's funny "GUI Bloopers"<sup>17</sup> (1999), David Kadavy's excellent "Design for Hackers"<sup>18</sup> (2011), and most importantly, with the review<sup>19</sup> in volume 40, issue 2 of the IEEE Annals of the History of Computing<sup>20</sup> of the work of Susan Kare<sup>21</sup>.

Cover photo by the author.

### REFERENCES

- <sup>1</sup> <https://www.linkedin.com/in/jenifertidwell/>
- <sup>2</sup> <https://www.oreilly.com/library/view/designing-interfaces-3rd/9781492051954/>
- <sup>3</sup> <https://dspace.mit.edu/handle/1721.1/14979>
- <sup>4</sup> [https://en.wikipedia.org/wiki/Douglas\\_Engelbart](https://en.wikipedia.org/wiki/Douglas_Engelbart)
- <sup>5</sup> <https://deprogrammaticaipsum.com/what-smalltalk-was-not/>
- <sup>6</sup> <https://deprogrammaticaipsum.com/jef-raskin/>
- <sup>7</sup> <https://deprogrammaticaipsum.com/issue/issue-001-hype/>
- <sup>8</sup> [https://en.wikipedia.org/wiki/Apple\\_Design\\_Awards](https://en.wikipedia.org/wiki/Apple_Design_Awards)
- <sup>9</sup> <https://deprogrammaticaipsum.com/kathy-sierra/>
- <sup>10</sup> [https://en.wikipedia.org/wiki/Brushed\\_metal\\_\(interface\)](https://en.wikipedia.org/wiki/Brushed_metal_(interface))
- <sup>11</sup> [https://en.wikipedia.org/wiki/Aqua\\_\(user\\_interface\)](https://en.wikipedia.org/wiki/Aqua_(user_interface))
- <sup>12</sup> [https://en.wikipedia.org/wiki/Windows\\_Aero](https://en.wikipedia.org/wiki/Windows_Aero)
- <sup>13</sup> [https://en.wikipedia.org/wiki/CSS\\_Zen\\_Garden](https://en.wikipedia.org/wiki/CSS_Zen_Garden)
- <sup>14</sup> <http://www.asymco.com/2019/05/16/the-pivot/>
- <sup>15</sup> <https://deprogrammaticaipsum.com/innovationscript/>
- <sup>16</sup> <https://deprogrammaticaipsum.com/issue/issue-036-innovation/>
- <sup>17</sup> <https://www.amazon.com/gp/product/1558605827/>
- <sup>18</sup> <https://www.amazon.com/Design-Hackers-Reverse-Engineering-Beauty-ebook/dp/B005J578EW>
- <sup>19</sup> <https://www.computer.org/csdl/magazine/an/2018/02/man2018020048/13rRUx0xPNH>
- <sup>20</sup> <https://www.computer.org/annals>
- <sup>21</sup> [https://en.wikipedia.org/wiki/Susan\\_Kare](https://en.wikipedia.org/wiki/Susan_Kare)

# Michael Feathers



By Graham Lee, August 1st, 2022

Adrian previously discussed *Working Effectively with Legacy Code* when he talked about how to choose a programming language for your book<sup>1</sup>. It deserves revisiting though, so here it is in the library section.

Quite simply, if you have not read this book yet, read it. If you have a colleague who has yet to read it, get them a copy. If someone asks you what one book to read about software engineering, it is this one. It is not *Code Complete, Second Edition*<sup>2</sup>, nor is it *Clean Code*, nor any other book that claims to teach you how to get software right the first time around (you will not). It is not *Programming Rust*, nor *Programming*

*Elixir*, nor any other book that claims to teach you a technology that solves all of your problems (it will not).

All of your code will either be abandoned or become legacy code, so if you do not plan on failing you should learn how to work effectively with legacy code.

The central premise to the book is that legacy code is usually not enjoyed because it is insufficiently tested. And that the way to work with it is to identify few, hopefully not too invasive, changes that allow for parts of the software to be tested in isolation. Once those parts are under test, it becomes easier to make more, perhaps more significant, changes to the parts that are tested to subdivide into smaller testable units. After enough iterations of this you have a system whose behaviour is both described and stabilised by the tests and will be easier to work with.

Why bother with all of this? For the same reason we avoid rewrites whenever a new programming language comes along<sup>3</sup>: the existing software encapsulates both the current behaviour of the software system, and the *desired* behaviour: whatever the software is supposed to do, people have adapted to whatever it actually *does* and so that must be a starting point for any future work.

It is all too easy to say “oh the legacy code is really buggy, we should start from scratch” but those bugs are the way the system—not just the software, but the socio-technical system in which it is embedded—works. The ones that have been fixed are hard-fought lessons about what people want from this software. It may not be well-designed—but it may be. What gets called bad design might actually be good design from a few years ago: software design is fad-led, rather than engineering-led. But it may also be the case that a clean design is hiding under the weight of various patches and hot fixes. This is exactly the situation that *Working Effectively with Legacy Code* will let you take control of: fixing the software towards a clean design without having to let go of the good, valuable behaviour. And of course in this age of Agile®©™, we work to the principle that the primary measure of progress is working software. The legacy software already works.

So why do software engineers prefer to start from scratch? Partly it is because programming is a monetised hobby<sup>4</sup>: people enjoy writing software so they will find ways to do that for income. But it is also a matter of capability. A Masters-

level course in software engineering<sup>5</sup> contains nothing about reading or adapting existing software; not even anything about buy-versus-build decisions. And most professional programmers are *not* trained software engineers. Without education or experience at reading, understanding, and modifying existing code, programmers see it as difficult, unnecessary effort. Why waste my time learning from person-centuries of experience at solving this problem, when I can type `cargo new` and have something that solves 5% of the problem badly in maybe a month or two?

And that is where this book comes in. It is your secret superpower. Learn how to work effectively with legacy code and you will be faster and more capable than almost all of the software writers working today, through this one weird trick: not writing most of the software you need.

Cover photo by Adrian Kosmaczewski.

### REFERENCES

<sup>1</sup> <https://deprogrammaticaipsum.com/how-to-choose-a-programming-language-for-your-book/>

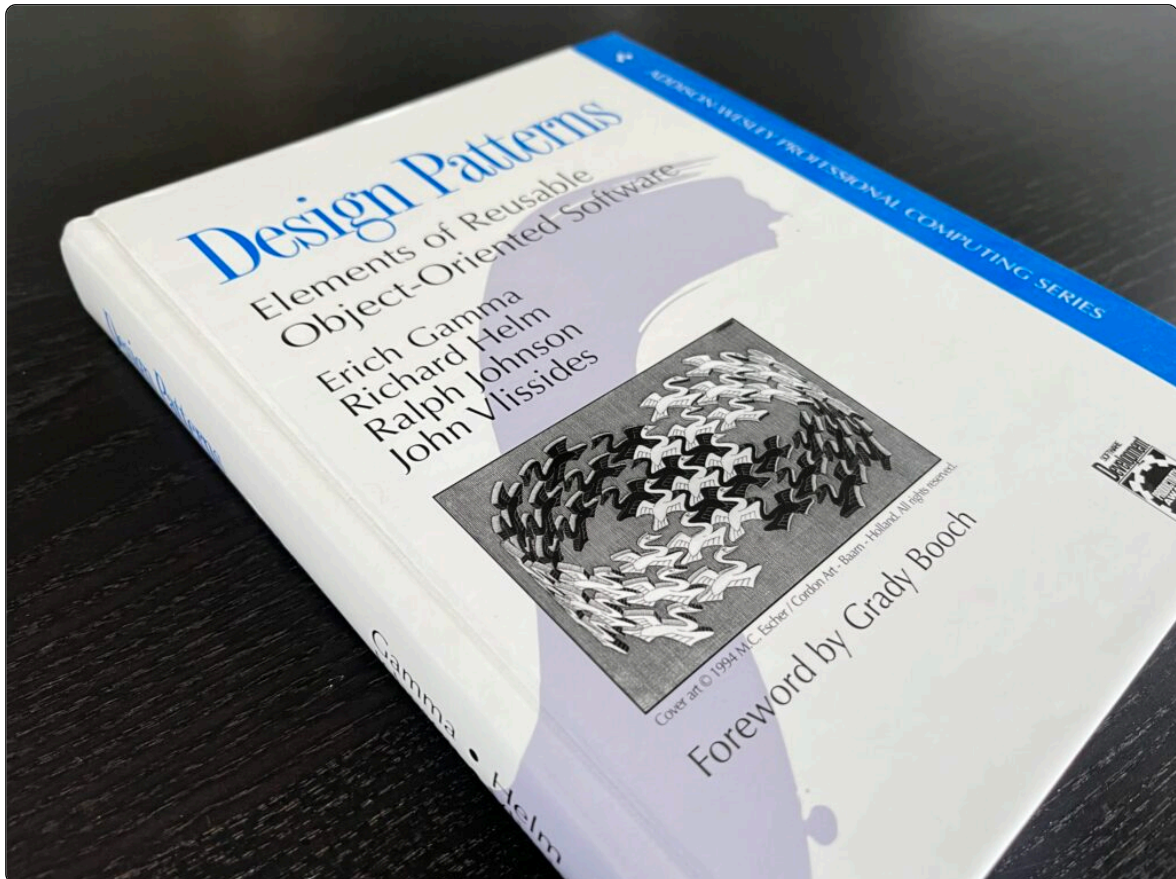
<sup>2</sup> <https://deprogrammaticaipsum.com/steve-mcconnell/>

<sup>3</sup> <https://deprogrammaticaipsum.com/the-double-denim-of-software-engineering/>

<sup>4</sup> <https://www.sicpers.info/podcast/episode-39-monetising-the-hobby/>

<sup>5</sup> <https://www.cs.ox.ac.uk/softeng/courses/subjects.html>

# The Gang Of Four



By Adrian Kosmaczewski, October 3rd, 2022

Many different things<sup>1</sup> bear the name “Gang of Four”; however, in this case, we are going to talk about a major bestseller in the history of computer books: “Design Patterns: Elements of Reusable Object-Oriented Software”<sup>2</sup> by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. There is a high probability that every reader of this article owns, has read, or has at least skimmed through the pages of a GoF book once or twice. The book has been reprinted dozens of times (40 times at least until 2012.) It has been the subject of uncountable articles, videos, panel discussions, and, yes, also attacks.

We are not going to delve into the book's contents, structure, or legacy; plenty of commentators have talked about that. Instead, we would like to focus on the various ways the book has been helpful to quite a few generations of computer scientists and software engineers. We will also not dive into the influence of Christopher Alexander's book "A Pattern Language"<sup>3</sup> on the Design Patterns movement. Lots have been said about that. Instead, we will point you to Mr. Alexander's talk at OOPSLA 1996<sup>4</sup>.

It is worth a mention (at least) that the GoF book dwarfed another excellent treatise on the subject, also named "Design Patterns for Object-Oriented Software Development,"<sup>5</sup> written by Professor Wolfgang Pree<sup>6</sup> and published the same year (1995) by the same publisher (Addison-Wesley) using code examples in the same language (C++.) Pree's book mentions Erich Gamma's 1991 thesis<sup>7</sup> at the University of Zürich as major influence and inspiration.

There are many reasons to think that the GoF book has been influential; after all, even if you do not like Stephen King novels or John Grisham novels, it is undeniable there are a lot of people who enjoyed them. So many, in fact, that Hollywood found on them quite a few successful blockbuster ideas. Yet, at the same time, purists will attack King and Grisham, arguing that they are neither Walt Whitman, Harper Lee, James Joyce, John Steinbeck, or Virginia Wolff. So be it; there is room for many different writers and writing styles.

We should apply the same criteria to computer books. We have already compared Donald Knuth to J. R. R. Tolkien<sup>8</sup>; let us try to find a similar analogy to "Design Patterns."

Structurally speaking, we can read this book in two movements; the first section contains some interestingly advanced (well, at least for 1995) observations on OOP analysis and design. The second section consists of the (in)famous catalog, describing 23 different design patterns. Instead, let us reshuffle the patterns and group them differently.

In our first category, we will place those patterns that proved to be very useful in many contexts: Visitor, Builder, Template Method, Observer, Decorator, Command, Adapter, Façade, and others. In the second group, those that did not: when

was the last time you applied the Flyweight, Memento, or Interpreter in your full-stack web application? Finally, there is a small group of eternal outcasts composed of Singleton, Abstract Factory, and Factory Method.

Now for the hard part: these groups are related to Stephen King's novel characters. Yes, you read right. We are pretty sure you did not see that one coming.

Each of the actually helpful OOP patterns in the first group reminds this author of Paul Sheldon, the unfortunate protagonist of "Misery."<sup>9</sup> They are somehow trapped by a group of crazy nurses called Java, C++, and C# (stuck in version 1.0), forcing them to generate lots of classes grouped in curious architectures, all while heavily sedated, unable to escape, and begging for help. At some point, a dynamically typed police force consisting of Ruby, Python, and JavaScript frees them from this ordeal.

The second group is akin to John Smith, the main clairvoyant character of "The Dead Zone."<sup>10</sup> These patterns are constantly aware of a horrendous possible future; they choose to act to prevent it from happening but disappear forever as soon as they jump into action.

Finally, the last group consisting of Singleton and the Factories are like "Carrie,"<sup>11</sup> mocked and bullied by a whole industry in a neverending prom, one which saw them covered with blood and laughed at every coronation ceremony. Yet, they persisted, and every so often, one of them reappears in our code and on our minds while they scream with rage and blow up our unit test suites with all their might in a catastrophic explosion of anger. Well, particularly Singleton, anyway.

Ironically enough, the Gang of Four was first humorously indicted of various sins<sup>12</sup> by their peers; this happened at OOPSLA 1999, where a "show trial"<sup>13</sup> was held in which the authors were found guilty. One of them, Richard Helm, confessed *in absentia*<sup>14</sup>, discharging his responsibility upon the other three accused.

Unfortunately, ten years after the book's publication, John Vlissides passed away<sup>15</sup> at 44, more or less at the same time when "Head First Design Patterns"<sup>16</sup> changed the game of computer books forever.

## BEST PRACTICES

Despite all the abuse the GoF book has suffered during the past quarter century, it remains a vital and utterly influential piece of work. Sure, we could replace many of these patterns with functional programming constructs, Aspect-Oriented Programming, or what-have-you paradigms<sup>17</sup>. Still, the careful depiction of each pattern in this colossal work as a coherent body of objects interacting at runtime has dramatically changed how we relate to software.

Whether we like it or not, design patterns give names to groups of objects; and Phil Karlton said<sup>18</sup>, naming is one of the hardest problems in computer science. We needed this taxonomy, and it proved helpful, if anything, to highlight the infinite flexibility of software and our minds.

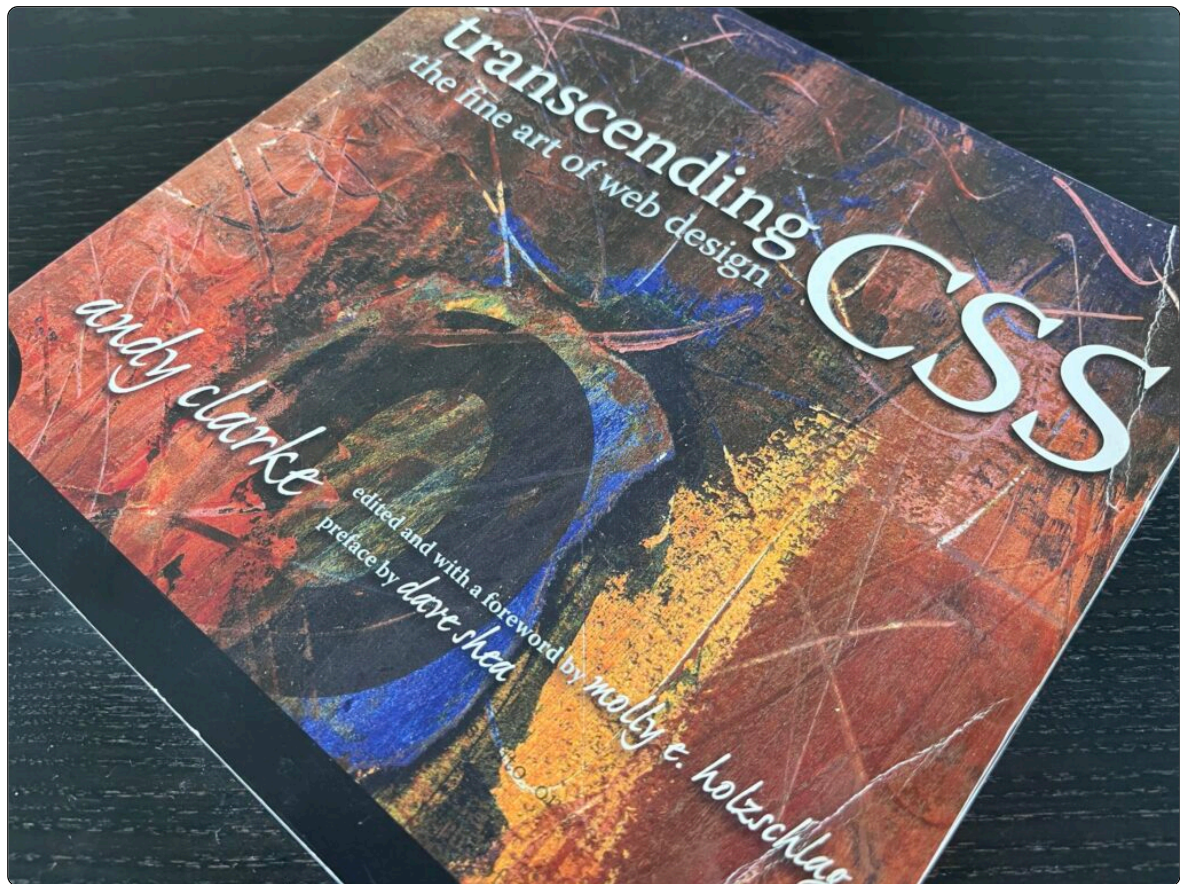
Cover photo by the author.

## REFERENCES

- <sup>1</sup> [https://en.wikipedia.org/wiki/Gang\\_of\\_Four\\_\(disambiguation\)](https://en.wikipedia.org/wiki/Gang_of_Four_(disambiguation))
- <sup>2</sup> [https://en.wikipedia.org/wiki/Design\\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns)
- <sup>3</sup> [https://en.wikipedia.org/wiki/A\\_Pattern\\_Language](https://en.wikipedia.org/wiki/A_Pattern_Language)
- <sup>4</sup> [https://www.youtube.com/watch?v=98LdFA-\\_zfA](https://www.youtube.com/watch?v=98LdFA-_zfA)
- <sup>5</sup> <https://archive.org/details/designpatternsfo00pree>
- <sup>6</sup> [https://en.wikipedia.org/wiki/Wolfgang\\_Pree](https://en.wikipedia.org/wiki/Wolfgang_Pree)
- <sup>7</sup> <https://link.springer.com/book/10.1007/978-3-642-77838-4>
- <sup>8</sup> <https://deprogrammaticaipsum.com/how-to-choose-a-programming-language-for-your-book/>
- <sup>9</sup> [https://en.wikipedia.org/wiki/Misery\\_\(novel\)](https://en.wikipedia.org/wiki/Misery_(novel))
- <sup>10</sup> [https://en.wikipedia.org/wiki/The\\_Dead\\_Zone\\_\(novel\)](https://en.wikipedia.org/wiki/The_Dead_Zone_(novel))
- <sup>11</sup> [https://en.wikipedia.org/wiki/Carrie\\_\(novel\)](https://en.wikipedia.org/wiki/Carrie_(novel))
- <sup>12</sup> <http://www.laputan.org/patterns/trial.html>
- <sup>13</sup> <http://www.laputan.org/patterns/gang-of-four.html>
- <sup>14</sup> <http://www.laputan.org/patterns/helm.html>
- <sup>15</sup> <https://www.washingtonpost.com/wp-dyn/content/article/2005/12/09/AR2005120902004.html>
- <sup>16</sup> <https://deprogrammaticaipsum.com/kathy-sierra/>
- <sup>17</sup> <https://deprogrammaticaipsum.com/alan-turing-wrote-object-oriented-code-in-c-and-ran-it-on-beam/>
- <sup>18</sup> <https://skeptics.stackexchange.com/a/39178>



# Andy Clarke



By Adrian Kosmaczewski, January 2nd, 2023

We have often said in the pages of this magazine that some books carry with them the Zeitgeist of their era. Examples are Bruce Tate’s “Beyond Java,” Joe Armstrong’s “Programming Erlang,” and Toby Segaran’s “Programming Collective Intelligence.” Such books have a tremendous impact upon publication, freezing in words not only a valuable body of knowledge, but also the spirit and promise of a new direction for the industry. Even if the APIs they describe become obsolete over time (which is mainly unavoidable), they remain as hallmarks of an

era, valuable witnesses of the preoccupations and needs of practitioners at the time of their publication.

This month we will review another of those books, “Transcending CSS: The Fine Art of Web Design”<sup>1</sup> by Andy Clarke<sup>2</sup>, published in 2006 by New Riders. Those were the times of Web 2.0, a rebirth from the ashes of the excesses of the dot-com boom of the previous decade. A moment to start anew, to apply the lessons learned, and to rebuild a new Web.

In those days, the state of the Web was not pretty; Internet Explorer 6 was already showing signs of decadence, yet it was far from being a priority for Ballmer’s Microsoft<sup>3</sup>. Its support for standards was mediocre at best, and upcoming products such as Apple Safari (whose WebKit rendering engine was a fork of KDE’s Konqueror<sup>4</sup>) and Mozilla Firefox (the most visible result of the “open-sourcing”<sup>5</sup> of Netscape’s code in 1998) were showing that alternatives were not only possible but also desirable.

Those were the times of Prototype.js<sup>6</sup>, script.aculo.us<sup>7</sup>, and jQuery. Jesse James Garrett had just given the Ajax name to “a new approach to web applications.”<sup>8</sup> Peter-Paul Koch (also known as “PPK”) was busy explaining how Quirks Mode<sup>9</sup> worked. New cross-platform JavaScript frameworks were helping developers work around the quirks of IE 6, unifying experiences and reducing development times. Frameworks like Ruby on Rails or Django were breaking waves, enabling unprecedented productivity and making the Web a fantastic place to make things again.

But in terms of design, who am I kidding; many of us still used the `<TABLE>` tag for the layout of our pages. I know it very well; I was one of those wannabe designers stuck in the past. In our defense, Your Honor, I will argue that those who started designing web experiences in 1997, like me, well, we did not have many tools at our disposal back then when Netscape 3.0 was still around. So, well, tables it was. *Mea culpa*: for a long time, CSS was, for me, just a fancy way to remove `<FONT>` tags and `ALIGN=` attributes on paragraphs. Ouch.

Fast-forward 10 years, and in 2007 the landscape was different and much more promising. We were at the point where CSS 3.0 was starting to become widely

available on browsers, which meant we could leave behind tables for layout altogether. And Andy Clarke's book was precisely the book that showed me how.

Andy was part of a larger group of web designers who started understanding the power of standards to make better websites. One of the most striking examples of that era was, without any doubt, Dave Shea's CSS Zen Garden. Maybe some of the readers of this piece will remember; it was a website whose premise was to showcase what CSS was capable of. Every link would load a different CSS stylesheet, always applied to the same standards-compliant HTML5 website. CSS Zen Garden is still online<sup>10</sup> at the time of this writing, and you can visit it just like fifteen years ago. Viewing the HTML source of the site shows a clean construction of intertwined `<DIV>`, `<UL>`, and `<SECTION>` tags, neatly woven into one another, describing the hierarchy and structure of a web page.

Not its look and feel; its structure. The core seed of progressive enhancement<sup>11</sup> as a design principle was laid out for the first time.

This is a fundamental concept: it felt like we were beginning to understand what this HTML thing was helpful for, *finally*. CSS was not only able to provide font-family or align information, but also to lay out those elements on a page, next to one another, on top of another, fixed on a specific position, or flowing as the user scrolled. The possibilities were endless.

Andy Clarke identified a significant problem with CSS, however. Designers are not developers, and CSS as *a language* was not a designer-friendly technology. He takes time to explain the meaning, the inner working, and the logic of CSS to an audience he knows very well; the book seems to be written for him, primarily, or at least for a younger version of himself. Such is the working progress that leads to many masterpieces, and this is no exception.

"Transcending CSS" is another work of art comparable to "Designing Interfaces" by Jenifer Tidwell, which we reviewed<sup>12</sup> precisely one year ago. The book is profusely illustrated and delightfully printed.

Dave Shea's work led to the 2005 book "The Zen of CSS Design"<sup>13</sup> written together with Molly Holzschlag... Molly edited and wrote the foreword to Andy's book, and Dave wrote the preface. We are all in good company here.

## BEST PRACTICES

The evolution of CSS did not stop after the book's release; far from that. "Transcending CSS" was released the same year as the iPhone, an invention that would put a fully-working web browser in people's pockets for the first time. In 2010, Ethan Marcotte<sup>14</sup> coined the term "Responsive Web Design"<sup>15</sup> to describe a technique consisting of using media queries and rules to select different stylesheets depending on the viewport size. The stage was set to create beautiful, flexible, and accessible user interfaces for any device, from the largest of TVs to the smallest of smartphones.

Andy Clarke revisited his book in 2019, and this new edition is available for everyone to read online<sup>16</sup>. If your work involves generating HTML content of any kind in any way, you owe it to yourself to take the time and dive into the mind of a man who can fight against King Kong<sup>17</sup> and save the world. And then read everything else<sup>18</sup> he has written; you will thank me later.

(And I promise I have not used a single `<TABLE>` to lay out any HTML page since I read this book.)

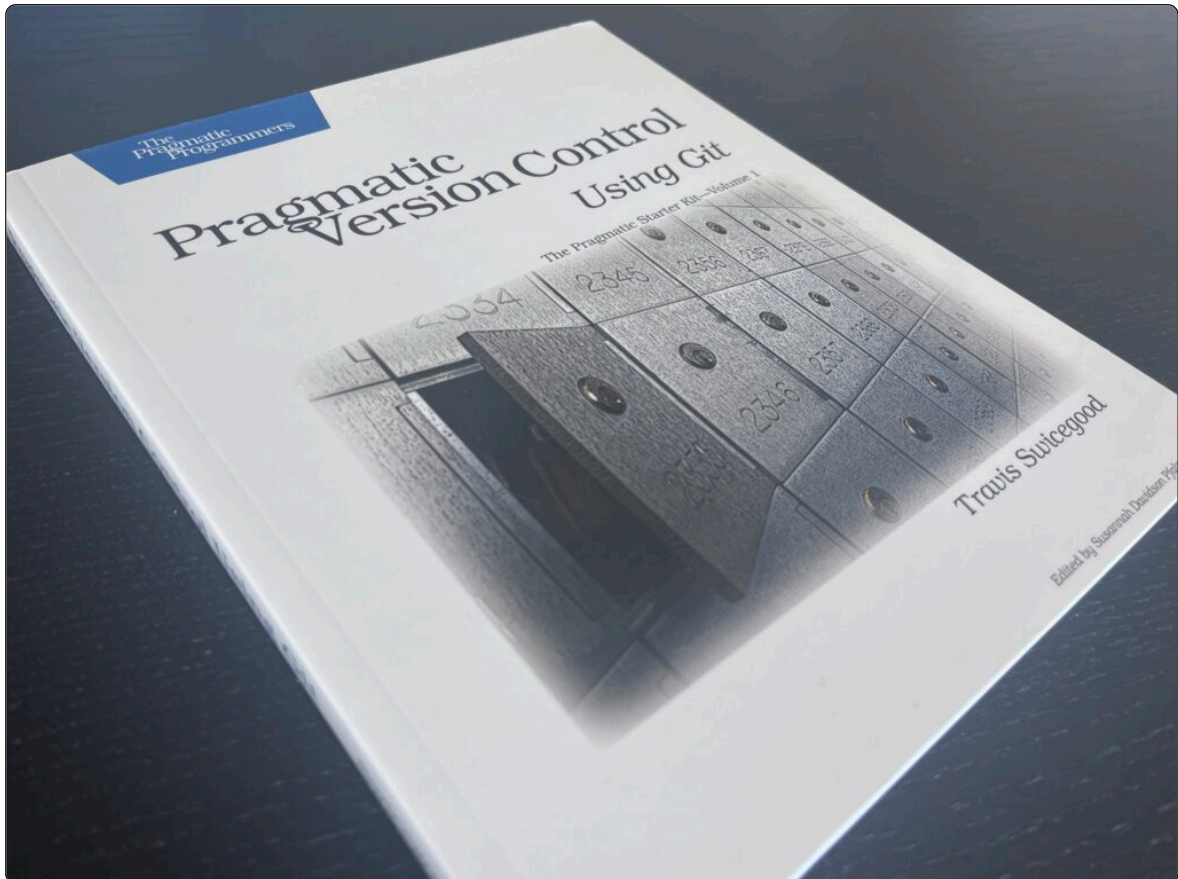
Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://www.peachpit.com/store/transcending-css-the-fine-art-of-web-design-9780321410979>
- <sup>2</sup> <https://stuffandnonsense.co.uk/>
- <sup>3</sup> <https://deprogrammaticaipsum.com/where-does-microsoft-want-to-go-today/>
- <sup>4</sup> <https://apps.kde.org/konqueror/>
- <sup>5</sup> <https://deprogrammaticaipsum.com/open-always-wins/>
- <sup>6</sup> <http://prototypejs.org/>
- <sup>7</sup> <http://script.aculo.us/>
- <sup>8</sup> <https://web.archive.org/web/20150910072359/http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>
- <sup>9</sup> <https://quirksmode.org/>
- <sup>10</sup> <https://www.csszengarden.com/>
- <sup>11</sup> [https://en.wikipedia.org/wiki/Progressive\\_enhancement](https://en.wikipedia.org/wiki/Progressive_enhancement)
- <sup>12</sup> <https://deprogrammaticaipsum.com/jenifer-tidwell/>
- <sup>13</sup> <https://www.amazon.com/exec/obidos/ASIN/0321303474/>
- <sup>14</sup> <https://ethanmarcotte.com/>
- <sup>15</sup> <https://alistapart.com/article/responsive-web-design/>
- <sup>16</sup> <https://stuffandnonsense.co.uk/transcending-css-revisited/index.html>
- <sup>17</sup> <https://mastodon.social/@malarkey/109466581363187302>
- <sup>18</sup> <https://stuffandnonsense.co.uk/books>



# Travis Swicegood



By Adrian Kosmaczewski, March 4th, 2024

A quick review of previous entries in the Library section of this magazine shows that it does not feature any book from The Pragmatic Programmers<sup>1</sup>, for no other reason than gross oversight. We have discussed books from MIT Press, Addison-Wesley, O’Reilly, and many other publishing houses, and now it is time to solve this issue. This month we will elaborate on “Pragmatic Version Control Using Git”<sup>2</sup>, a 2008 book by Travis Swicegood<sup>3</sup>.

In other circles than software, such as physics or literature, the year 2008 would undoubtedly be called an *annus mirabilis*. While the “real world” was grappling

with one of the most severe financial crisis<sup>4</sup> in history, the software industry saw a miraculous conjunction of factors, seemingly happening within months of one another. Let us enumerate a few: in March, the availability of the iPhone SDK. In April, the launch of GitHub. And in September, three major events: the announcement of the Google Chrome web browser on the 2nd, the launch of Stack Overflow on the 15th, and the release of Android 1.0 on the 23rd.

Couple all this with the meteoric rise in popularity of AWS and cloud computing in general, the phenomenon of social media, and the explosion of Web 2.0 and “Ajax”<sup>5</sup>, with the ubiquitous presence of libraries such as jQuery<sup>6</sup> and Prototype.js<sup>7</sup>. You could feel that software engineering was never going to be the same from that point onwards. It was undergoing a profound transformation, live, in front of our eyes.

It was both a terrific and terrible time to be a software developer, and Git appeared as one of the hottest new tools to check out. But Git was also substantially different from anything many of us had used up to that point. At least for me, the whole idea of a distributed source code management system was completely alien.

Up to that moment I had been a happy Subversion<sup>8</sup> user, hosting some of my code on Google Code<sup>9</sup> and even running Subversion repository servers on my own machine—I admit, a rather clumsy setup for a single developer. Before Subversion, I had used various source code management tools: Rational ClearCase<sup>10</sup> from 2002 to 2003, Visual SourceSafe<sup>11</sup> from 2004 to 2005, and CVS<sup>12</sup> from 2006 to 2007. (Yes, the list shows a clear regression; it seemed that the further I advanced in my career, the dumber and more unstable the SCM of my employer would be. *Anyway.*)

All the systems enumerated above are client-server, and thus, centralized; there is a server, somewhere, and a small client on my laptop allows me to check code out, commit changes, and voilà. Need to blame a file? Talk to the server. Want to see the full history log? Talk to the server. Want to create a branch? You get the gist.

I remember attending a talk about Git during a community meetup in March 2008, and rushing back home to order a book about it on Amazon (Kids: e-books were just starting to appear on our radar, as the original Kindle<sup>13</sup> was launched in

November 2007. So, physical book it was.) There were not many books published about Git (yet) and I had always enjoyed reading books from The Pragmatic Programmers, so choosing this month's Library book was a no-brainer for me.

Needless to say, this was the perfect book to open my mind to the possibilities of Git. Just like in many other Pragmatic Bookshelf titles, the introduction to the subjects is gentle, complete, yet entertaining. The first chapter was particularly helpful, with its explanation of the differences between distributed and centralized version control, and the resulting workflows that were possible in each case.

The conceptual part was the most challenging one for me, but the author made it abundantly clear that it was not *that* hard. And the highly practical nature of the book, filled with examples, made understanding Git fast, straightforward, and simple. The detachable "Git Command Quick Reference" sat on my desk for months, providing a much-needed reference until most commands were engraved in my mind.

The rest is history. When I started my own business<sup>14</sup> in 2008, I got a paying GitHub account and used it to host the private Git repositories of all of my customers. But some of those customers were not entirely used to Git or distributed source code management for that matter, and asked me to push code into their Subversion repositories; no problem at all, `git svn` to the rescue<sup>15</sup>.

Git was a godsend during those years of independent consulting. I was very frequently on the go, either on a plane or on a train, without any network connectivity, and Git allowed me to work perfectly well, committing, diff'ing, blaming (mostly me) and branching as much as I needed, at any time, in any place. As soon as a network connection became available, a simple `git push` would share those changes with my customers, and even better, trigger some automated task in a CI/CD system somewhere.

Beyond my own developer experience, one thing that changed in the software industry from 2008 onwards was that Git became the all-encompassing source code management tool, while others slowly disappeared into oblivion. New developers joining the workforce learned Git exclusively. New businesses started adopting Git and collaborating on GitHub or GitLab repositories. Popular software projects

## BEST PRACTICES

started opening up shop on GitHub. Text editors started providing built-in Git support. Everyone started assuming that Git was the big default SCM of the world of software. Well-known SCM packages slowly faded away. Companies providing commercial SCMs went bankrupt or pivoted to other markets.

During the past 16 years, I almost exclusively used Git for my code and my projects, both personal and professional. I did use Mercurial too, for a while at least, but for no other reason than my employer at the time was a heavy Python<sup>16</sup> user. They ended up migrating to Git at some point, the power of networking effects being too difficult to counter. Those forces even bent BitBucket<sup>17</sup>, originally a Django web application hosting only Mercurial repositories, to drop Mercurial support in 2020. *Sign o' the times.*

The writing was on the wall. Git is the *de facto* standard, and it has become as ubiquitous as the English language, or water. Travis Swicegood's book still sits on my bookshelf; admittedly, I do not consult it so often as I did in those early days, but it represents a pivotal moment for my career—and truly, for the whole industry.

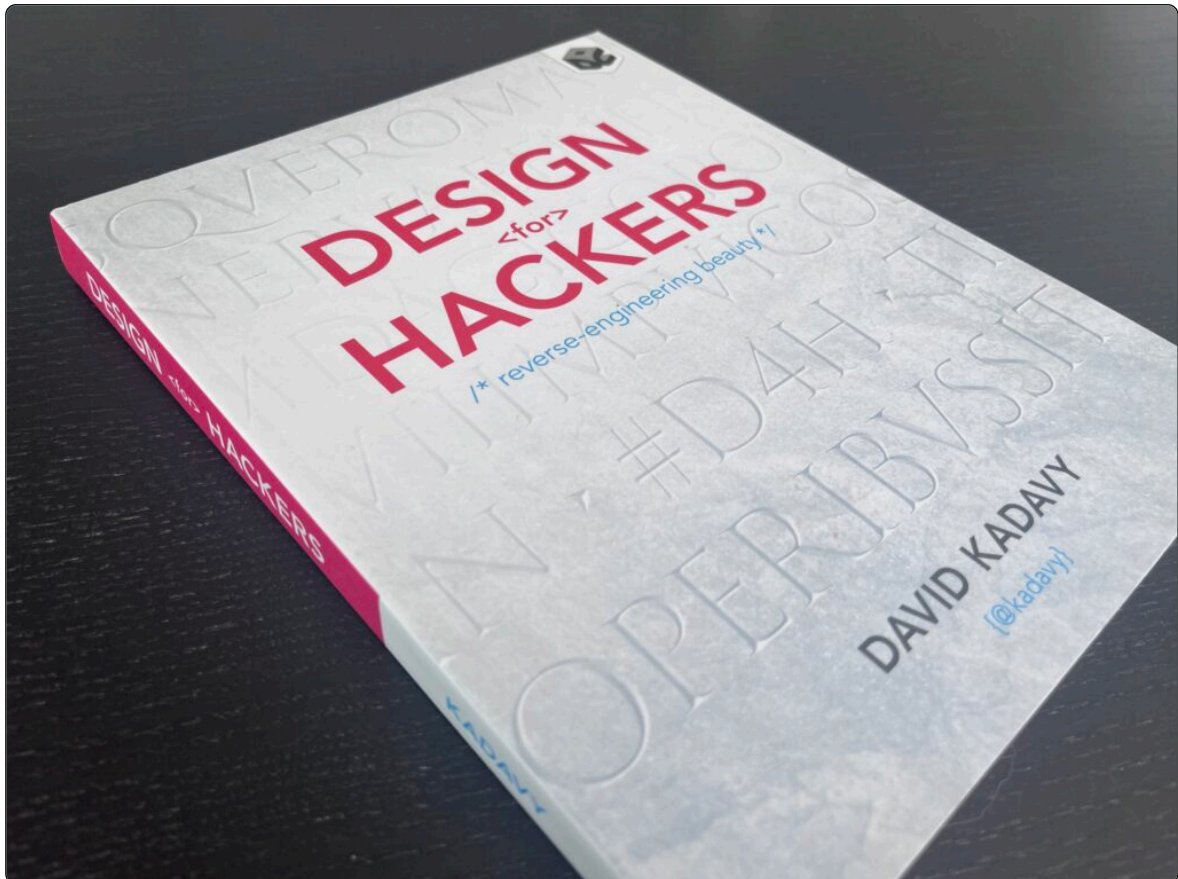
Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://pragprog.com/>
- <sup>2</sup> <https://pragprog.com/titles/tsgit/pragmatic-version-control-using-git/>
- <sup>3</sup> <https://github.com/tswicegood>
- <sup>4</sup> [https://en.wikipedia.org/wiki/2007-2008\\_financial\\_crisis](https://en.wikipedia.org/wiki/2007-2008_financial_crisis)
- <sup>5</sup> [https://en.wikipedia.org/wiki/Ajax\\_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))
- <sup>6</sup> <https://en.wikipedia.org/wiki/JQuery>
- <sup>7</sup> <http://prototypejs.org/>
- <sup>8</sup> <https://subversion.apache.org/>
- <sup>9</sup> <https://code.google.com/>
- <sup>10</sup> [https://en.wikipedia.org/wiki/Rational\\_ClearCase](https://en.wikipedia.org/wiki/Rational_ClearCase)
- <sup>11</sup> [https://en.wikipedia.org/wiki/Microsoft\\_Visual\\_SourceSafe](https://en.wikipedia.org/wiki/Microsoft_Visual_SourceSafe)
- <sup>12</sup> [https://en.wikipedia.org/wiki/Concurrent\\_Versions\\_System](https://en.wikipedia.org/wiki/Concurrent_Versions_System)
- <sup>13</sup> [https://en.wikipedia.org/wiki/Amazon\\_Kindle](https://en.wikipedia.org/wiki/Amazon_Kindle)
- <sup>14</sup> <https://akos.ma/blog/running-akosma-software/>
- <sup>15</sup> <https://git-scm.com/docs/git-svn>
- <sup>16</sup> <https://deprogrammaticaipsum.com/the-state-of-python-in-2021/>
- <sup>17</sup> <https://en.wikipedia.org/wiki/Bitbucket>



# David Kadavy



By Adrian Kosmaczewski, May 6th, 2024

Few ecosystems react as viscerally and as brutally to “bad” visual design than whatever Apple has brought into the world. From the first Macintosh to the latest Vision Pro, the whole idea of making apps for Apple platforms involves, hopefully sooner than later, a severe and serious evaluation of style along with functionality.

Let us look at some examples of when said ecosystem reacted viscerally and brutally. On October 3rd, 2009, Brandon Pittman published an article titled “The Worst Twitter Client... Ever” on the Smoking Apples website, nowadays available<sup>1</sup> thanks

to the commendable work of the Internet Archive. Here are some quotes from this gem of an article:

*If there was ever a reason Steve Jobs didn't want people making apps for iPhone, ChillTwit was it. (...) I honestly believe the SA editors assigned me this review to see if I'd kill myself. (...)*

*I can't find one redeeming quality about this app. (...) the fact that he's trying to get \$0.99 out of people pisses me to no end. (...)*

*(...) if you buy this, we're not friends anymore.*

You get the idea.

In the same vein, one year earlier, John Gruber, author of the influential Daring Fireball blog, published a picture on his Flickr account<sup>2</sup> (we cannot get any more retro than this!). The caption reads:

*A screenshot from Stevens Creek Software's upcoming iPhone app, TripLog/1040. I'm not even sure where to start.*

There is a direct connection between Pittman's and Gruber's criticisms (and the discussion<sup>3</sup> it generated in the latter case) with MKBHD's recent review of the Humane AI Pin<sup>4</sup> and the controversy<sup>5</sup> surrounding it. But as usual, I digress. The important point here is that, for some platforms like iOS, looks matter more than software developers would like to admit, and reviews can make or break entire product lines (and even businesses) in no time.

Of course, not all ecosystems display such levels of vitriol against amateurish looks. In no particular order, Windows, Android, and Linux in general, are environments where functionality, price, and stability will always be primed over good looks. This is fine and good, but where it becomes problematic is where users and developers on these platforms deride or make fun of a supposedly "lesser" world such as Apple's.

Again, hubris and disdain does not help anyone. Some users prefer beautiful things, some others prefer working software, and some others prefer both at the same time.

Faced with the need to publish applications that work well and look decently good, the question software developers ask themselves is, then, how can you learn good visual design when all you have is a computer science degree? Turns out that a designer from Silicon Valley not only answered this question, but also created a fundamental piece of work in the process.

Published in 2011, “Design for Hackers” by David Kadavy dives into the common elements of design: typography<sup>6</sup>, composition, color, and starts a much-needed discussion about why design is important, and how technology both influences and is influenced by it.

It is profusely illustrated, and the tagline “reverse-engineering beauty” gives a good hint of the contents and the intended audience.

“Design for Hackers” can work as a study and reference material, including some appendixes at the end with useful typography tips and tricks, for example about how to pair fonts, or how to use them properly in various contexts.

This work belongs to my favorite kind of reading: a treatise that bridges the gap between two worlds that display impossible dialogues<sup>7</sup> and have a tendency to clash for the pettiest of reasons. Designers and engineers often coexist in software development teams in a false dichotomy, angry at each other, blaming each other, instead of trying to understand each other-which, spoiler alert, is the most important thing you can do with fellow humans, at all times.

Proportion, whitespace, hierarchy, rhythm, are all words that have different meanings in the minds of software developers, and to be able to have a conversation with the designers in your team, you would be wise to learn them.

Inversely, I can only recommend visual designers to learn more about the constraints and limits of software and hardware, and to get acquainted with the design guidelines of your chosen platform. I cannot tell how many times I have had to explain to designers those limits, trying to make them understand the tradeoffs

in cost and speed of development their teams could reach by following them. Once again, the biggest issue blocking teams from delivering good software is communication.

For those software developers interested in the art and science of visual design, I could recommend some other important works. First, the quintessential “The Design of Everyday Things” by Don Norman<sup>8</sup>. We have talked in the pages of this magazine about “Designing Interfaces” by Jenifer Tidwell<sup>9</sup> (about desktop software design) and “Transcending CSS” by Andy Clarke<sup>10</sup> (about web app design), and we repeat the recommendation now. Finally, take a look at “Graphic Design: The New Basics” by Ellen Lupton and Jennifer Cole Phillips<sup>11</sup> for a complete overview of basic concepts.

If you look to deepen your knowledge of visual design with additional information, head over to [books.design](#)<sup>12</sup>, an incredible resource created<sup>13</sup> by Norma, a “non-graphic design studio in Turin”. Finally, for a social perspective of the impact of design in our modern world, “Design is a Job”<sup>14</sup> by Mike Monteiro (of who we have already talked about<sup>15</sup> in this magazine) is a mandatory reading. Also noteworthy, the newsletter “Not a Designer”<sup>16</sup>.

Making better software does not only mean making applications with good modularity, plenty of unit tests, and many useful features, but also with better looks, decent levels of usability, and with enough attention paid to accessibility. These are indeed useful qualities, and will help you to, at least, avoid scathing reviews of your next app.

Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://web.archive.org/web/20091006005807/http://smokingapples.com/iphone/app-store-iphone/the-worst-twitter-client-ever>
- <sup>2</sup> <https://flickr.com/photos/gruber/2635257578>
- <sup>3</sup> <https://daringfireball.net/linked/2008/07/09/triplog-redux>
- <sup>4</sup> <https://www.youtube.com/watch?v=TitZV6k8zfA>
- <sup>5</sup> <https://www.youtube.com/watch?v=QztFpzKsdeA>
- <sup>6</sup> <https://deprogrammaticaipsum.com/gary-hustwit/>
- <sup>7</sup> <https://deprogrammaticaipsum.com/the-impossible-dialogue-revisited/>
- <sup>8</sup> <https://mitpress.mit.edu/9780262525671/the-design-of-everyday-things/>
- <sup>9</sup> <https://deprogrammaticaipsum.com/jenifer-tidwell/>
- <sup>10</sup> <https://deprogrammaticaipsum.com/andy-clarke/>
- <sup>11</sup> <https://ellenlupton.com/Graphic-Design-The-New-Basics>
- <sup>12</sup> <https://books.design/>
- <sup>13</sup> <https://normadesign.it/en/projects/books-by-designers/>
- <sup>14</sup> <https://www.designisajob.com/>
- <sup>15</sup> <https://deprogrammaticaipsum.com/mike-monteiro/>
- <sup>16</sup> <https://notadesigner.io/>



# Amy Brown & Greg Wilson



By Adrian Kosmaczewski, June 3rd, 2024

Most software developers are ejected from academia into the jaws of the business of software with little preparation. Of course, they are equipped with good enough knowledge about some more or less relevant programming language, and maybe some algorithm, hopefully including the venerable linked list reversion, indispensable to pass the dreaded coding interview. But not much more.

Of course, the availability of free and open-source code, coupled with the rallying cry of Reddit users “read the source, Luke!” means that they should be able to pick up major ideas just by cloning the repository of their preferred software package, and peruse its source code while sipping their preferred caffeinated drink.

The reality, needless to say, is far more complicated. Source code merely does a good enough job at the transcription of algorithms (in the best cases), but it does a terrible one at conveying the nature of the high-level architecture that organizes and coordinates all of that code in memory once the application is running.

This proverbial lack of visibility of the “blueprint” of any software artifact is one of the reasons why “Software Architects” have such an important role in teams dealing with large source code bases, where “large” is defined in average as more than a hundred thousand lines of code (a threshold very easily accessible, I am afraid).

We need, then, some documentation about the architecture of the code. But, alas, documentation is not always available, not because Johnny cannot code<sup>1</sup>, but because he does not want<sup>2</sup> to document his work.

This is where the two tomes of “The Architecture of Open Source Applications” edited by Amy Brown and Greg Wilson shine. They provide a much-needed explanation of the structure and inner workings of major products available in the free and open-source software market, written by their authors or maintainers themselves. After all, who could better describe their architectures?

Put together, this work contains 49 chapters, each describing various kinds of applications: desktop utilities such as the Eclipse IDE<sup>3</sup> or Audacity<sup>4</sup>; source control systems like Git<sup>5</sup> or Mercurial<sup>6</sup>; server-side products like nginx<sup>7</sup>, Moodle<sup>8</sup>, or Riak<sup>9</sup>; or developer tools like CMake<sup>10</sup>, LLVM<sup>11</sup>, or ZeroMQ<sup>12</sup>. And much more.

Perhaps surprisingly, each chapter follows a different path; the architecture required to organize the code of each of these packages is, needless to say, wildly different from one another. Each one solves a particular problem, and the spirit behind the solution will vary accordingly.

Do not expect these chapters to be a “quick guide” to understand how to architect (is there such a verb?) your own project; these texts must be understood as individual use cases, each tackling a very defined set of issues, and each in their own way. You will have to come up with the architecture of your own projects, maybe starting with a wise decision and applying Conway’s Law<sup>13</sup> before writing a single line of code.

And for that, these are truly inspiring chapters, including some background details about the size of the teams, the longevity of the codebase, and anecdotes about their evolution. The social context around the source code is pervasive, and it is an integral part of each project. One cannot separate the existence of a software project from the social structures underpinning its architecture. Both coexist and shape each other continuously.

That is, by far, the most important takeaway of these essays, and the most salient factor that readers should remark while reading these pages. Of course, the technical details are juicy; but they are not, despite the enthusiasm of each author and most readers, the most important part. The social networks built around free and open-source projects are a defining aspect; alas, they are another characteristic not reflected at all in whatever programming language chosen by the authors.

The two volumes of “The Architecture of Open Source Applications” are available for free online<sup>14</sup>, under a Creative Commons Attribution 3.0 Unported<sup>15</sup> license. But if you enjoy the content, you should also buy a paperback or digital copy, knowing that the editors are donating all royalties to Amnesty International<sup>16</sup>. And to make the proposal even more enticing, the website includes the full text of two other gems: “500 Lines or Less” edited by Michael DiBernardo, and “The Performance of Open Source Applications” edited by Tavish Armstrong.

As stated on the website:

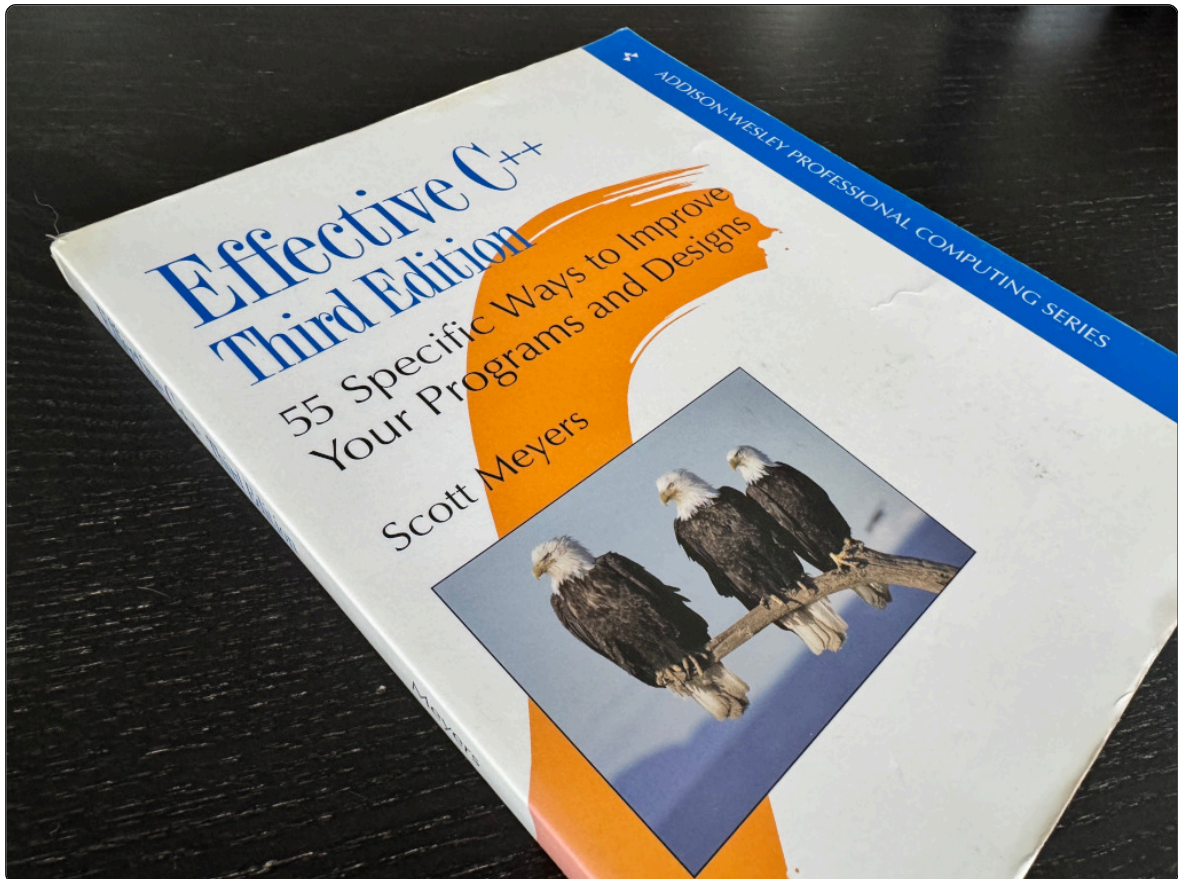
*If you are a junior developer, and want to learn how your more experienced colleagues think, these books are the place to start. If you are an intermediate or senior developer, and want to see how your peers have solved hard design problems, these books can help you too.*

Cover photo from the book website.

### REFERENCES

- <sup>1</sup> [https://www.salon.com/2006/09/14/basic\\_2/](https://www.salon.com/2006/09/14/basic_2/)
- <sup>2</sup> <https://deprogrammaticaipsum.com/on-the-aversion-to-writing/>
- <sup>3</sup> <https://www.eclipse.org/downloads/>
- <sup>4</sup> <https://www.audacityteam.org/>
- <sup>5</sup> <https://git-scm.com/>
- <sup>6</sup> <https://www.mercurial-scm.org/>
- <sup>7</sup> <https://nginx.org/en/>
- <sup>8</sup> <https://moodle.org/>
- <sup>9</sup> <https://riak.com/>
- <sup>10</sup> <https://cmake.org/>
- <sup>11</sup> <https://llvm.org/>
- <sup>12</sup> <https://zeromq.org/>
- <sup>13</sup> [https://en.wikipedia.org/wiki/Conway's\\_law](https://en.wikipedia.org/wiki/Conway's_law)
- <sup>14</sup> <https://aosabook.org/en/>
- <sup>15</sup> <http://creativecommons.org/licenses/by/3.0/legalcode>
- <sup>16</sup> <http://amnesty.org>

# Scott Meyers



By Adrian Kosmaczewski, February 3rd, 2025

Around 20 years ago I found a job as a C++ developer. My new employer provided me the first day with a PDF file defining the very strict and mandatory set of guidelines to be followed for the production of code in the organization. These rules can be summarized as follows: do not use the Standard Template Library; do not use templates; and do not use multiple inheritance. If you are a C++ developer reading the previous phrase, I hope you can understand the dismay I felt while reading that. If you are not a C++ developer, suffice to say that to this day I do not understand why would anyone choose to use C++ *without* those features.

A programming language is called a “language” for a reason: it contains a set of grammar rules and a dictionary of reserved words, which can be put together in certain ways to produce a particular semantic. No French poet would consciously avoid Alexandrines<sup>1</sup> other than for stylistic purposes, and no Latin poets stayed away from Saturnians<sup>2</sup> when the time came. Those forms are tools, and even if admittedly, human languages are much less mathematical than programming ones, they are all part of a poet’s arsenal, and rightfully so.

I have long meditated about the reasons behind the conscious choice of forbidding certain forms of C++ in this company’s product. I have concluded that it had less to do with the explicit reasons they gave in their guidelines document (readability, maintainability, some other ability) and more with the fact that in 2006 C++ was stuck in a limbo. The standard in vogue at the time, the 2003 edition, which was a bug-correcting revision of the 1999 one, was quickly losing its prominence as an application programming language, replaced by other, more hyped, languages, almost all at a higher level in the abstraction hierarchy, and featuring a proverbial garbage collector: Java<sup>3</sup>, C#, Scala, Erlang, Ruby, and Python<sup>4</sup>.

(It must be said at this point, that the software product in question was a business application written for Windows, whose development had originally started in the mid-1990s, and which used object-oriented programming features extensively to model various business entities.)

We must remember that 2006 was also the year of Bruce Tate’s “Beyond Java”<sup>5</sup>; a time of experimentation and possibilities. JavaScript had all of a sudden “good parts”<sup>6</sup>; Rails was taking everyone by surprise. The OOP Hype Cycle<sup>7</sup> was getting stuck in its trough of disillusionment. Last but not least, it was also the time of “Writing Secure Code”<sup>8</sup> being a mandatory reading at Microsoft, with the public opinion openly blaming C and C++ and their goddamn pointers for pretty much every software security disaster making headlines.

(To make a long story short, my employer chose to move to Java shortly after, for the next version of their flagship application. This was a rewrite that proved costly, not only in financial but also, and painfully enough, in human terms. But that is material for another story.)

Back to the coding guidelines, the explicit choices therein (no multiple inheritance, no STL, no templates) made the codebase unwieldy, heavy, prone to repeated code, and hard to evolve. In particular, avoiding multiple inheritance even prevented the team from using pure abstract classes with virtual methods (akin to Java and C# interfaces, or Objective-C and Swift protocols) which are, by all means, a fantastic abstraction and design mechanism. I could digress for many more pages; in short, and in hindsight, I do consider such choices a sad mistake.

During the year and a half I spent in that company, however, I had the immense chance of discovering the outstanding work of Scott Meyers<sup>9</sup>. To say that his book “Effective C++: 55 Specific Ways to Improve Your Programs and Designs” had a strong effect in me is an understatement. I still own my copy of the second edition (published in 2005), and to this day, even though C++ has evolved in ways that nobody could have predicted back in 2006 (and thankfully so!) it still remains a staple of my library, and the *go-to* book whenever I have to work on C++ code (which is not that often, alas). Last but not least, this was the book that opened the door to my university degree<sup>10</sup> in 2008.

Scott (I hope he does not mind me calling him by his first name) is one of my all-time heroes in the world of programming. To give you an idea why, I will just quote a 2003 panel where he was asked about how to interview programmers<sup>11</sup>, and his observations are golden:

*I hate anything that asks me to design on the spot. That's asking to demonstrate a skill rarely required on the job in a high-stress environment, where it is difficult for a candidate to accurately prove their abilities. I think it's fundamentally an unfair thing to request of a candidate.*

Yes, Scott. This magazine wholeheartedly<sup>12</sup> agrees<sup>13</sup> with you.

But back to the book. There is one precise reason why I admire (and still consult) “Effective C++” 20 years after its publication; C++ might have changed in the past two decades, yet the tenets that define the spirit of the language (most of which are described in the book) are still the same. I have grown a fondness for boring technology, proven time and again, and backed by a solid ecosystem around it: C++ ticks all of these boxes.

“Effective C++” is, in essence, a book quite similar to Robert L. Glass<sup>14</sup> “Facts and Fallacies of Software Engineering”: a series of very specific guidelines, stating clearly what to do (and most importantly, what *not* to do) in various situations and conundrums. In this case, regarding the essence of C++.

Let us see some examples, starting with Item 1, “View C++ as a federation of languages”:

*Today’s C++ is a multiparadigm programming language, one supporting a combination of procedural, object-oriented, functional, generic, and metaprogramming features.*

Yes, functional features, too. C++ is a very complex beast, offering unprecedented levels of flexibility and power. Why restrict oneself?

The power of C++ often comes hidden, however, and as Item 5 explains, developers must “Know what functions C++ silently writes and calls”.

*When is an empty class not an empty class? When C++ gets through with it. If you don’t declare them yourself, compilers will declare their own versions of a copy constructor, a copy assignment operator, and a destructor.*

Developers new to C++ are simply unaware of such rules<sup>15</sup>; they are certainly not obvious at first sight, and can lead them to write code that simply does not behave as expected. In the same vein, Item 7 reminds you of making destructors virtual in polymorphic class families, while Item 9 tells you not to call virtual functions in constructors nor destructors.

Scott dives into the famous RAI<sup>16</sup> principle in the third chapter, starting with Item 13, “Use objects to manage resources”. Then it uses chapters 4 and 6 to explain code design techniques, arguably useful not only for C++ code but in so many other languages. The quintessential example? “Item 22: Declare data members private”. Duh.

Item 31, “Minimize compilation dependencies between files” has the intended effect of speeding up the compilation of C++ projects, a fact often neglected in

large codebases. For the anecdote, the system of my employer consisted of half a million lines of code, and took around three hours to compile on a standard single-core PC of 2006, using the current version of the Microsoft C++ compiler. One of my colleagues literally spent a weekend applying Scott's best practices, and cut down the compilation time... in half.

Finally, Item 53 is a pet peeve of mine: "Pay attention to compiler warnings". Although to be honest, I would have not only suggested to pay attention to them, but literally to remove them altogether. I also remember applying this principle to Objective-C code<sup>17</sup> back when I was earning a living as an iPhone app developer. (In general, I considered<sup>18</sup> knowing C and C++ as a very important part of the upbringing of pre-Swift iOS developers.)

In hindsight, I have the impression that giving a copy of Scott Meyers' "Effective C++" to each one of my colleagues would have been a better choice than penning an overly restrictive *ad hoc* document. These days, the C++ Core Guidelines<sup>19</sup> live document by none other than Stroustrup and Herb Sutter<sup>20</sup> (the last update of which, at the time of this writing, happened October last year) took over the role that "Effective C++" played 20 years ago, becoming the most authoritative coding guidelines document available today.

Scott decided to retire from the C++ world in 2015<sup>21</sup>, so let this article be a belated "thank you" from a developer who benefitted tremendously from his outstanding contributions to the field. We need more people like him.

If you want to continue your exploration of C++, I can recommend the eponymous book<sup>22</sup> by Stroustrup himself, or if you are in a hurry, its shorter version<sup>23</sup>. A decidedly more advanced reading, Andrei Alexandrescu's "Modern C++ Design: Generic Programming and Design Patterns Applied"<sup>24</sup> is one of the most extensive works on template metaprogramming ever written, a book highly recommended by Scott Meyers himself. The first chapter of the (often-mentioned in the pages of this magazine) book "Masterminds of Programming"<sup>25</sup> by Federico Biancuzzi and Shane Warden contains an extensive interview of Bjarne Stroustrup.

If you are interested in the genesis and history of the language, you will be happy to learn that C++ is the *only* programming language to have been

featured in three consecutive editions of the famous “History of Programming Languages” (HOPL) conferences, created by the late Jean Sammet<sup>26</sup>. The three papers, all written and delivered by Stroustrup himself, are available online: “A history of C++: 1979–1991”<sup>27</sup> (HOPL II, 1993), “Evolving a language in and for the real world: C++ 1991-2006”<sup>28</sup> (HOPL III, 2007), and “Thriving in a crowded and changing world: C++ 2006–2020”<sup>29</sup> (HOPL IV, 2021). At the current rate, HOPL V will take place in 2035, so you have time to read them all.

C++ is a complex yet rewarding language, and it is time for software developers to understand that it has changed for good, that it has all the traits of a “modern” language, and that it definitely deserves a place in the programming arsenal of each one of us in this industry.

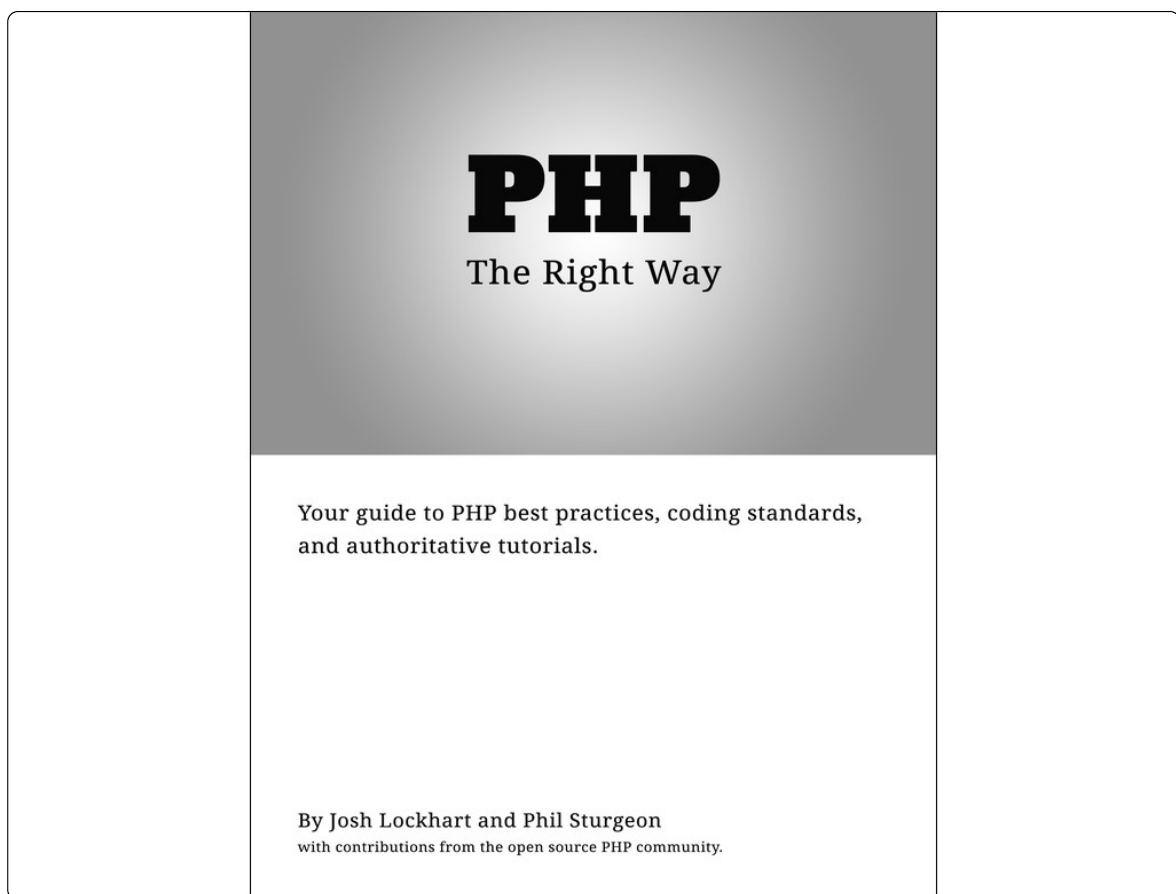
Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://en.wikipedia.org/wiki/Alexandrine>
- <sup>2</sup> [https://en.wikipedia.org/wiki/Saturnian\\_\(poetry\)](https://en.wikipedia.org/wiki/Saturnian_(poetry))
- <sup>3</sup> <https://deprogrammaticaipsum.com/java-the-programmer-environment-that-has-it-all/>
- <sup>4</sup> <https://deprogrammaticaipsum.com/the-state-of-python-in-2021/>
- <sup>5</sup> <https://www.oreilly.com/library/view/beyond-java/0596100949/>
- <sup>6</sup> <https://deprogrammaticaipsum.com/douglas-crockford/>
- <sup>7</sup> <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- <sup>8</sup> <https://deprogrammaticaipsum.com/microsofts-writings-on-security/>
- <sup>9</sup> [https://en.wikipedia.org/wiki/Scott\\_Meyers](https://en.wikipedia.org/wiki/Scott_Meyers)
- <sup>10</sup> <https://akos.ma/blog/master/>
- <sup>11</sup> <https://www.artima.com/articles/how-to-interview-a-programmer>
- <sup>12</sup> <https://deprogrammaticaipsum.com/do-not-ask-me-about-how-interviewing-works/>
- <sup>13</sup> <https://deprogrammaticaipsum.com/tales-of-the-interview/>
- <sup>14</sup> <https://deprogrammaticaipsum.com/robert-l-glass/>
- <sup>15</sup> [https://en.wikipedia.org/wiki/Rule\\_of\\_three\\_%28C%2B%2B\\_programming%29](https://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29)
- <sup>16</sup> [https://en.wikipedia.org/wiki/Resource\\_acquisition\\_is\\_initialization](https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization)
- <sup>17</sup> <https://akos.ma/blog/objective-c-compiler-warnings/>
- <sup>18</sup> <https://akos.ma/blog/how-knowing-c-and-c-can-help-you-write-better-iphone-apps-part-1/>
- <sup>19</sup> <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- <sup>20</sup> <https://deprogrammaticaipsum.com/herb-sutter/>
- <sup>21</sup> <http://scottmeyers.blogspot.com/2015/12/good-to-go.html>
- <sup>22</sup> <https://www.stroustrup.com/4th.html>
- <sup>23</sup> <https://www.stroustrup.com/tour3.html>
- <sup>24</sup> [https://en.wikipedia.org/wiki/Modern\\_C%2B%2B\\_Design](https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design)
- <sup>25</sup> <https://www.oreilly.com/pub/pr/2277>
- <sup>26</sup> <https://deprogrammaticaipsum.com/jean-sammet/>
- <sup>27</sup> <https://dl.acm.org/doi/10.1145/154766.155375>
- <sup>28</sup> <https://dl.acm.org/doi/10.1145/1238844.1238848>
- <sup>29</sup> <https://dl.acm.org/doi/10.1145/3386320>



# Josh Lockhart & Phil Sturgeon



By Adrian Kosmaczewski, August 4th, 2025

The infinite flexibility of software is not without some major disadvantages. That is the main reason why we, software practitioners, crave so much any kind of information about “the best” or “the right” way to build, test, deploy, and maintain our systems. Yes, our craft is already complicated enough, and we are not even talking about the human complexities like office layouts, employment shenanigans, dress

codes, and whatnot. In this occasion we are going to talk about a resource that fights, with facts and examples, the battle of excellency in the world of PHP.

In the pages of this “Library” section we have often covered major books aiming to provide such guidance; for example Michael Feathers’ “Working Effectively with Legacy Code”<sup>1</sup>; Scott Meyers’ “Effective C++”<sup>2</sup>; Amy Brown & Greg Wilson’s “Architecture of Open Source Applications”<sup>3</sup>; Jenifer Tidwell’s “Designing Interfaces”<sup>4</sup>; David Kadavy’s “Design for Hackers”<sup>5</sup>; Steve McConnell’s “Code Complete”<sup>6</sup>; Douglas Crockford’s “JavaScript: The Good Parts”<sup>7</sup>; and others. So many, actually, that we have added them all to a new category in this blog, called “Best Practices”<sup>8</sup>, which you can access at any time through the main menu.

Of course, this month’s entry, Josh Lockhart & Phil Sturgeon’s excellent “PHP: The Right Way” falls into this category. Solely mentioning the names of these two authors, however, I am acutely aware of the implicit, tacit, long list of contributors that have provided corrections, translations, reviews, and more to this work. Said list even includes some university professor like Kris Jordan<sup>9</sup>. But please understand that not enumerating them all here does not, in any way, diminish the relevance of their contributions. Let us just say that Josh and Phil are the instigators of this book, and not just the initial authors.

(And knowledgeable initial instigators they are, oh yes: Josh is the author of “Modern PHP”<sup>10</sup>, a book published by O’Reilly in 2015, and the creator of the Slim Framework<sup>11</sup>, a favorite choice of mine in the world of PHP frameworks. Phil has worked as a consultant for various tech companies, but most importantly, he is behind the Protect Earth<sup>12</sup> project, a commendable reforestation effort we can only salute and support.)

As it stands, “PHP: The Right Way”, currently available on Leanpub<sup>13</sup> in PDF or EPUB formats, and hosted on its own website<sup>14</sup> for instant access, is the fruit of an open-source project hosted on GitHub<sup>15</sup> with (at the time of this publication) more than 300 collaborators.

Now that is what I call a community.

Why read this? Well, to put it bluntly, this book aims to mark a milestone, leaving some classic books behind, like Rasmus Lerdorf’s own “PHP Pocket Reference”<sup>16</sup>

published by O'Reilly in 2000 (you can still read an excerpt from it on the Internet Archive<sup>17</sup>, by the way) or Paul Hudson's "PHP in a Nutshell"<sup>18</sup> from 2005 (is it the same Paul Hudson that nowadays hacks with Swift<sup>19</sup>?) That "old PHP" is the default impression most developers have, particularly those who might have worked with the language and ecosystem decades ago, not realizing 23 years later that both had (thankfully) evolved for the better. Time for a wake-up call, if you will.

This book aims to point you towards what is considered the "State of the Art" and "Crème de la Crème" of tooling, practices, and patterns for your PHP code in 2025. From testing, to caching, to dependency management, to OOP, to internationalization and localization, and even to containerization, you will find enough information to help you along the way. This is a live document, updated every so often since its first version in 2013, translated to 20 languages other than English, and continuously evolving. Which, let us be honest, also explains why there is no printed version available—and rightfully so.

(The value you could derive from "PHP: The Right Way" assumes, of course, that you have not fallen prey of the latest fads around "vibe coding"<sup>20</sup>, in which case, well, good luck with that. The whole premise of this magazine is very simple: that you care about your craft enough as to understand what is going on behind the scenes of the code that you put into production. That is all.)

PHP in 2025 is a fabulous beast with a thriving ecosystem, powering most of what you see on the Internet. This month's Library book, "PHP: The Right Way" is precisely the most appropriate choice to either discover *or* to get re-acquainted with PHP after a long hiatus.

We recommend coupling the lecture of this book with more resources. Books such as the 2023 "PHP Cookbook"<sup>21</sup> by Eric A. Mann. Online resources, like "Learn modern PHP"<sup>22</sup> by the prolific Daniel Opitz<sup>23</sup>, or "Clean Code concepts adapted for PHP"<sup>24</sup> by Piotr Plenik. And modern best practices documents and manifestos, such as "The Twelve-Factor App"<sup>25</sup>, "The Reactive Manifesto"<sup>26</sup>, "Rootless Containers"<sup>27</sup>, "Keep a Changelog"<sup>28</sup>, "Oh Shit, Git!?"<sup>29</sup>, "Reproducible Builds"<sup>30</sup>, and the always useful "Bobby Tables: A guide to preventing SQL injection"<sup>31</sup>.

## BEST PRACTICES

Oh, and if you excuse the shameless plug, and as mentioned previously, many of the books in the “Best Practices”<sup>32</sup> category of this magazine could also be a welcome lecture in your road to betterment.

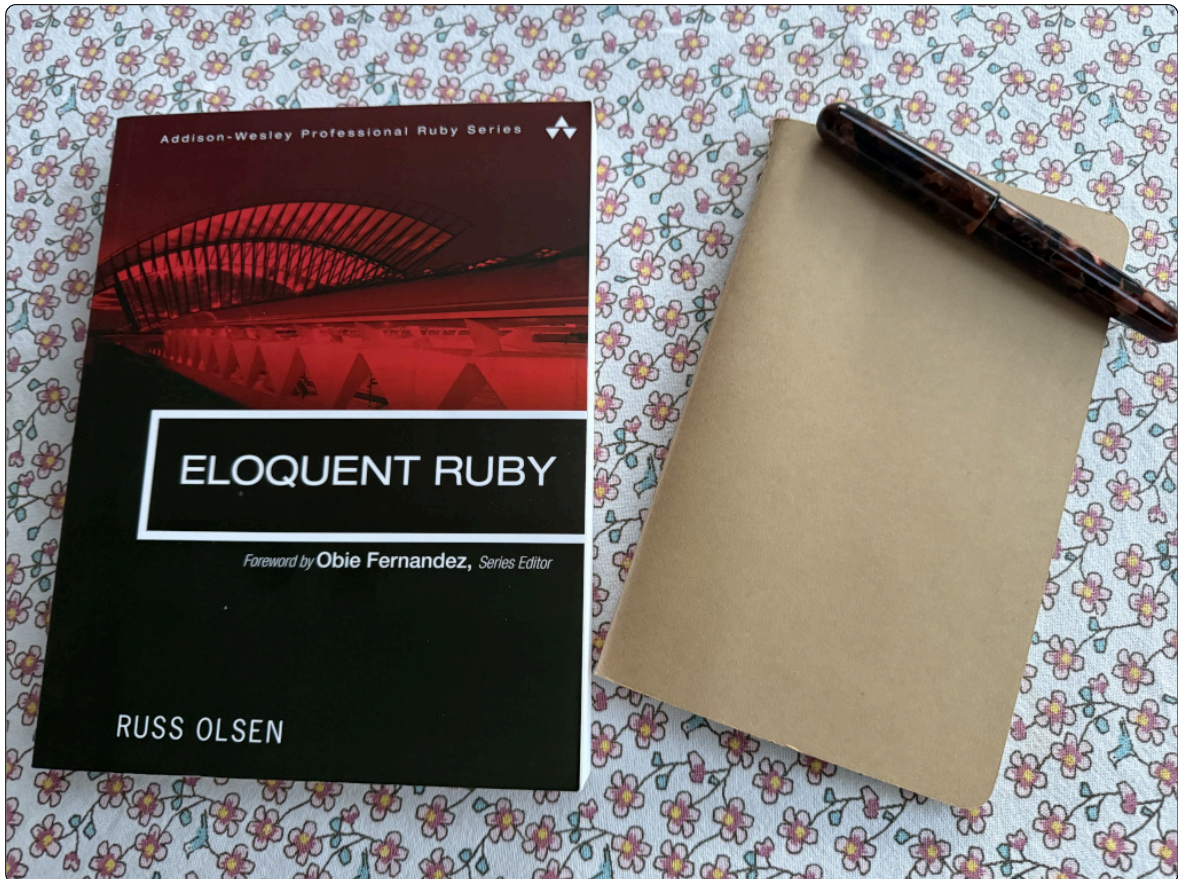
Cover photo adapted from the EPUB file.

## REFERENCES

- <sup>1</sup> <https://deprogrammaticaipsum.com/michael-feathers/>
- <sup>2</sup> <https://deprogrammaticaipsum.com/scott-meyers/>
- <sup>3</sup> <https://deprogrammaticaipsum.com/amy-brown-greg-wilson/>
- <sup>4</sup> <https://deprogrammaticaipsum.com/jenifer-tidwell/>
- <sup>5</sup> <https://deprogrammaticaipsum.com/david-kadavy/>
- <sup>6</sup> <https://deprogrammaticaipsum.com/steve-mcconnell/>
- <sup>7</sup> <https://deprogrammaticaipsum.com/douglas-crockford/>
- <sup>8</sup> <https://deprogrammaticaipsum.com/category/best-practices/>
- <sup>9</sup> <https://krisjordan.com/>
- <sup>10</sup> <https://www.oreilly.com/library/view/modern-php/9781491905173/>
- <sup>11</sup> <https://www.slimframework.com/>
- <sup>12</sup> <https://www.protect.earth/>
- <sup>13</sup> <https://leanpub.com/phptherightway>
- <sup>14</sup> <https://phptherightway.com>
- <sup>15</sup> <https://github.com/codeguy/php-the-right-way>
- <sup>16</sup> <https://app.oreilly.com/pub/pr/879>
- <sup>17</sup> [https://web.archive.org/web/20000815221550/http://www.oreilly.com/catalog/phppr/chapter/php\\_pkt.html](https://web.archive.org/web/20000815221550/http://www.oreilly.com/catalog/phppr/chapter/php_pkt.html)
- <sup>18</sup> [https://www.goodreads.com/book/show/43241.PHP\\_in\\_a\\_Nutshell](https://www.goodreads.com/book/show/43241.PHP_in_a_Nutshell)
- <sup>19</sup> <https://www.hackingwithswift.com/>
- <sup>20</sup> <https://deprogrammaticaipsum.com/the-allure-of-vibe-coding/>
- <sup>21</sup> <https://www.oreilly.com/library/view/php-cookbook/9781098121310/>
- <sup>22</sup> <https://odan.github.io/learn-php/>
- <sup>23</sup> <https://odan.github.io/about.html>
- <sup>24</sup> <https://github.com/piotrplenik/clean-code-php>
- <sup>25</sup> <https://12factor.net/>
- <sup>26</sup> <https://www.reactivemanifesto.org/>
- <sup>27</sup> <https://rootlesscontaine.rs/>
- <sup>28</sup> <https://keepachangelog.com/en/1.1.0/>
- <sup>29</sup> <https://ohshitgit.com/>
- <sup>30</sup> <https://reproducible-builds.org/>
- <sup>31</sup> <https://bobby-tables.com/>
- <sup>32</sup> <https://deprogrammaticaipsum.com/category/best-practices/>



# Russ Olsen



By Graham Lee, February 2nd, 2026

Programming languages have settled into a comfortable middle age, with most coming to resemble one another. Many use some variety of ALGOL-derived structure, maybe with the occasional feature that first made its appearance in a LISP variant. Execution starts at the beginning of the file, or the entry point function. Each statement runs in sequence — except the ones with the magic keywords that say something else happens. Edsger Dijkstra would be perfectly at home using any of these languages, with the cognitive toolbox he developed in the 1960s. So why do they look so *different*?

Mostly, it is because communities of practice develop around these programming languages, and these communities have different rules. Étienne Wenger<sup>1</sup> came up with the idea of a community of practice<sup>2</sup> to describe the observation that people who share a passion come together to perform their shared passion, and get better through these interactions. Such communities develop distinct identities, based on the way that they communicate how they do what they do — and also on what they do not communicate. Tacit knowledge — the unspoken norms in a community — acts as a kind of reinforcement of the strength of membership. It is not necessarily gatekeeping for a community to have tacit knowledge; these are not the secret mysteries of some shadowy guild. They are things that nobody bothers mentioning because everybody thinks that everybody else already knows them.

As such, what we learn from looking at a different programming language in the 2020s is not so much a different way of *programming*, as it is a different *way* of programming. We learn what the community around that language values, the tricks they think are clever (and the tricks they think overly clever), and the ways of using their tools that communicate membership in the community.

As Adrian explored in our C++ issue, “Effective C++”<sup>3</sup> is a description of the family of distinct programming languages that comprise C++, and guidance of which of those you should be using. Other programming galaxies provide similar material: “Effective Java”<sup>4</sup> tells you which bits of the Java language to avoid. Famously, “JavaScript: The Good Parts”<sup>5</sup> is a fraction of the size of a definitive guide to the language. This book very clearly delineates the “bad” areas of JavaScript, devoting much of its slim space to telling you how to stop remembering that they exist.

Two programming galaxies take a different approach: they take great delight in showing you all the features the language has on offer, and the fun you can have by trying them out. Perl is one, and Ruby is the other.

A guide to Java would probably steer you away from the Reflection API. A Swift book would tell you that its runtime metaprogramming capabilities are mostly there for backward compatibility. “Eloquent Ruby”<sup>6</sup>, a book by Russ Olsen, dedicates seven chapters to metaprogramming and invites you to fill your boots.

We start, however, with the chapter that tells you what the rules of engagement for the community of practice are: *Write Code That Looks Like Ruby*. Throughout this chapter, we receive guidance to create code that is “readable”; of course, implicitly this means “readable to people who like to read code that looks like this”. Readable to members of the community of practice. Readable to Rubyists.

The remaining chapters give guidance on using the various features of Ruby: strings and symbols; objects and classes (of course, as Everything Is An Object™ in Ruby); the regular expression API; and the aforementioned metaprogramming facilities. Each chapter contains about a page on what you might see “in the wild”, helping to contextualize its advice or identify alternative ways of doing the same thing. It also contains somewhere between a paragraph and a couple of pages on “staying out of trouble”: guidance to avoid problems caused by, for example, using a symbol where you need a string, or vice versa.

It is here that we see the difference between the programming galaxies. Guidance on C++, Java, JavaScript, and friends tells you which bits of the language to avoid in order to get your work done. Guidance like “Eloquent Ruby” shows you what the language is capable of, and waits to see what creative things you produce when you make use of them.

Cover photo by the author.

### REFERENCES

<sup>1</sup> [https://en.wikipedia.org/wiki/%C3%89tienne\\_Wenger](https://en.wikipedia.org/wiki/%C3%89tienne_Wenger)

<sup>2</sup> <https://www.wenger-trayner.com/introduction-to-communities-of-practice/>

<sup>3</sup> <https://deprogrammaticaipsum.com/scott-meyers/>

<sup>4</sup> <https://www.oreilly.com/library/view/effective-java-3rd/9780134686097/>

<sup>5</sup> <https://deprogrammaticaipsum.com/douglas-crockford/>

<sup>6</sup> <https://www.oreilly.com/library/view/eloquent-ruby/9780321700308/>