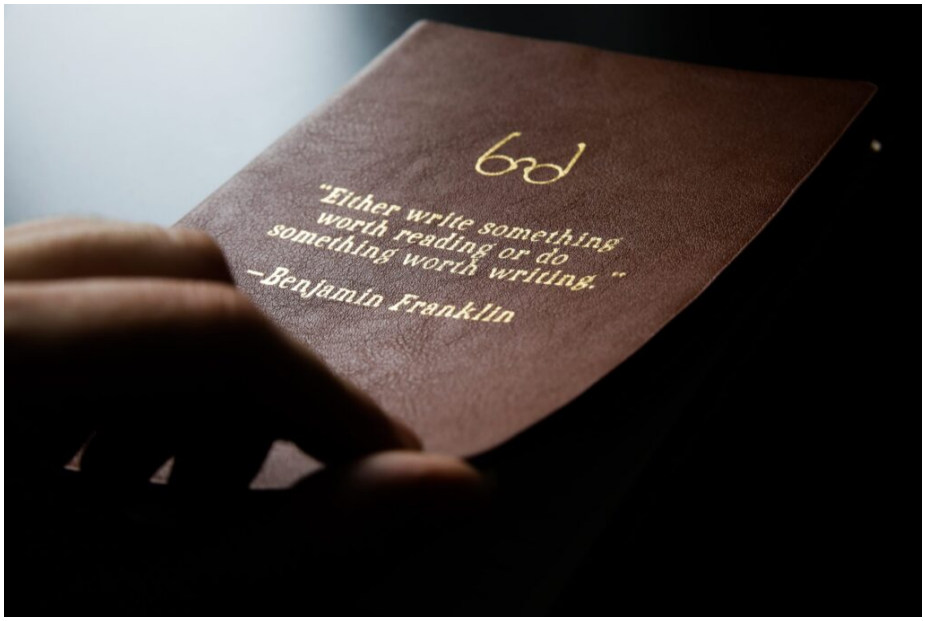
Issue 015: Writing

Adrian Kosmaczewski

December 2nd, 2019



Welcome to the fifteenth issue of *De Programmatica Ipsum*, dedicated to the subject of *Writing*. In this edition we will start a new section, called "Library[1]", where we will discuss the best books ever written in the field of software engineering:

- Graham enumerates all the valid reasons any team should write more prose[2] than code.
- Adrian tries to find out why software teams do not write[3] documentation.
- Graham inaugurates the Library section with a review of two of Brad Cox's major works[4]: *Object-Oriented Programming: an Evolutionary Approach* and *Superdistribution*.

Enjoy this issue! Please subscribe to our free newsletter[5] to stay updated about new releases.

Cover photo by Mona Eendra[6] on Unsplash[7].

[1] https://deprogrammaticaipsum.com/category/library/
[2] https://deprogrammaticaipsum.com/why-my-team-writes/
[3] https://deprogrammaticaipsum.com/on-the-aversion-to-writing/
[4] https://deprogrammaticaipsum.com/brad-cox/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://unsplash.com/@monaeendra?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[7] https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Why My Team Writes

Graham Lee

December 2nd, 2019

Writing is one of the most disruptive technologies ever invented after agriculture. Before writing, people could pass information in the current moment, to those who were close enough to hear noises made by other people. Writing allows people to communicate across space, passing written information to readers without the physical presence of the author. And it allows them to communicate across time, leaving evidence of thoughts long after the thinker had died.

As software engineers, our job is basically to wrangle information and make it useful. We might as well be called "informaticians"—and in many languages, we would be. The information we work with takes multiple forms.

Information About Problems

Sometimes we call them requirements, sometimes stories, sometimes domain knowledge. We often need to record information about how the world works, or how people work with the world, or how people work with each other in the world. This information helps us to

identify problems people have, that we might be able to solve using computers.

It also helps us to identify *people who have those problems*, or "potential customers". Someone who can see that we're thinking about their problem can consider engaging with us to determine whether our solution is valuable.

Information About Solutions

My colleagues and I need to agree on what the thing we're building *does*, and how it does it. We draw box-and-arrow diagrams showing our thinking, we write READMEs telling each other how to set things up, and comments and wiki pages describing particularly complex parts of the solution or hard-won nuggets of domain knowledge that have made their way into the code.

We write messages in our automated tests explaining what it means to the app if a particular assertion fails. We do the same thing in assertion expressions in the code, too. We name things to help people understand what they do.

And, of course, we tell those people who *have the problem* what our solution is, how it works, and how they can use it. This is the part that turns them from potential customers into actual customers. Or into not customers, with reasons for not being customers that are useful information in themselves.

Information About Technology

We make cases for using particular tools. We explain tricky things about the tools we are using. We write training materials for people who want to learn about the tools we're using. We write blog posts describing our experiences with those technologies. We make records of decisions that we made, and alternatives we rejected.

Information About People

We document our expectations of interactions with each other. We reflect on how our work has proceeded, and we write about what we'd like to change. We write about what we're planning to work on next, and whether we're stuck on any of it. We write descriptions of the people we want to work with, questions to ask to discover their suitability, and evaluations of their answers.

Cover photo by Florencia Viadana[1] on Unsplash[2].

[1] https://unsplash.com/@florenciaviadana?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[2] https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

On The Aversion To Writing

Adrian Kosmaczewski

December 2nd, 2019

Around 90% of the teams I have worked with in the past 22 years have never, ever, documented anything. Not a single wiki page, not a README file on top of a repository, not a single PDF file for the end users, not even a single UML diagram. Where they successful? Hardly.

Such dysfunctional and dystopian organizations, regularly release software without any kind of written documentation for human beings. The excuses for this situation range from the pitiful ("documentation is always out of sync with the code") to the infurating ("engineers are not good at writing anyway") to the downright outrageous ("the code is self-documenting.") As my friend Graham explained recently[1], code is *not* the final product of a software engineering team; the final product is the solution to a particular problem, which incidentally, happens to be provided as code.

In a world of Turing-complete languages, the choice of the programming tool used to generate that solution boils down to irrational personal preferences and economic constraints, only. In such a point of view, which the author of these lines has irrevocably and adamantly sustained for decades, documentation is a *conditio sine qua non* for the final solution of said problem.

Let us be very clear in this point. There is, and there cannot be, any software without human-readable, documentation.

Having said that, it is commonplace to observe that for some inner reason, most software developers have an innate, deep, absolute hatred of using keyboards for anything else than

[1] https://twitter.com/iwasleeg/status/1200432280600227847

cranking out code. Can one blame them? Not really. The truth is, the whole education system is built in such a way that it is a *de facto* writing hater factory.

Kids are told to write dissertations and later papers, a dry and lonely activity. Those papers are graded with red pens, filled with scribbles, with illegible or otherwise undecipherable comments, and more dissertations and papers follow, one after the other.

And then one day, before they graduate, they are told to write a thesis, around 14'000 words. The mere thought of counting words, even if automated, adds weight to an already dreadful exercise. There is even the legitimate concern that some automated software will flag those texts, as a false positive case of collusion or plagiarism.

Writing is an act of nervousness and tension, certainly not pleasure.

And then there is the problem of tooling. Most people relate the word "writing" with Microsoft Word, which is a sad association. They do not only dread the act of writing, but they came to abhor the tool they must use to pour their thoughts into. And even worse, they might as well have witnessed the thing crashing down, taking to oblivion the effort of the past few hours of work. Some others have similar experiences with tools like Atlassian Confluence or other wiki engines, and with some ill-conceived tablet applications.

The problem is when the whole act of writing, unfortunately associated with deficient tooling, is sent to the garbage bin in a single sweeping gesture.

No wonder apps like WriteRoom[2], iA Writer[3], and Ulysses[4], became so popular among writers in the past decade. No wonder why the latest Freewrite[5] sold out. If you decide to write seriously, you actually need a tool that will work with you, not against you.

I am not here to advocate for the use of Asciidoctor[6], Markdown[7], or even LaTeX[8] (although I will if you ask me.) But I would rather have teams write text in Notepad than in any other tool. I would rather have software teams write something at all, at least a couple of paragraphs a day, describing their current work, their current designs, their current statuses.

Besides, the idea of all of these tools is actually the same: to remove the friction between the brain and the fingers. All modern computers already understand plain text *per se* (arguably, UTF-8 is one of the greatest inventions of all times,) and come usually bundled with a keyboard; you do not need anything else to transform your thoughts into a digital form. And the consensus among writers is, indeed, that dropping Microsoft Word is the first step towards writing happiness.

The healthiest teams, the most productive teams I have ever worked with, wrote everything down. In excruciating detail. They even had some ISO certification which actually forced them to have everything written down. They even had "peer writing," "extreme writing," and "mob writing" sessions, all together collaborating on some document simultaneously. Writing documentation was considered an integral part of their delivery process, not an afterthought, not a nice-to-have, not the job of somebody else.

But, of course, there is a first step to actually start pouring those words on paper. Almost

[2] http://www.hogbaysoftware.com/products/writeroom
[3] https://ia.net/writer
[4] https://ulysses.app/
[5] https://getfreewrite.com/
[6] https://asciidoctor.org/
[7] https://daringfireball.net/projects/markdown/
[8] https://www.latex-project.org/

20 years ago, Joel Spolsky famously said[9] that "writing is a muscle." So close this browser window, and start writing about your software.

Cover photo by Danielle MacInnes[10] on Unsplash[11].

[9] https://www.joelonsoftware.com/2000/10/02/painless-functional-specifications-part-1-why-bother/

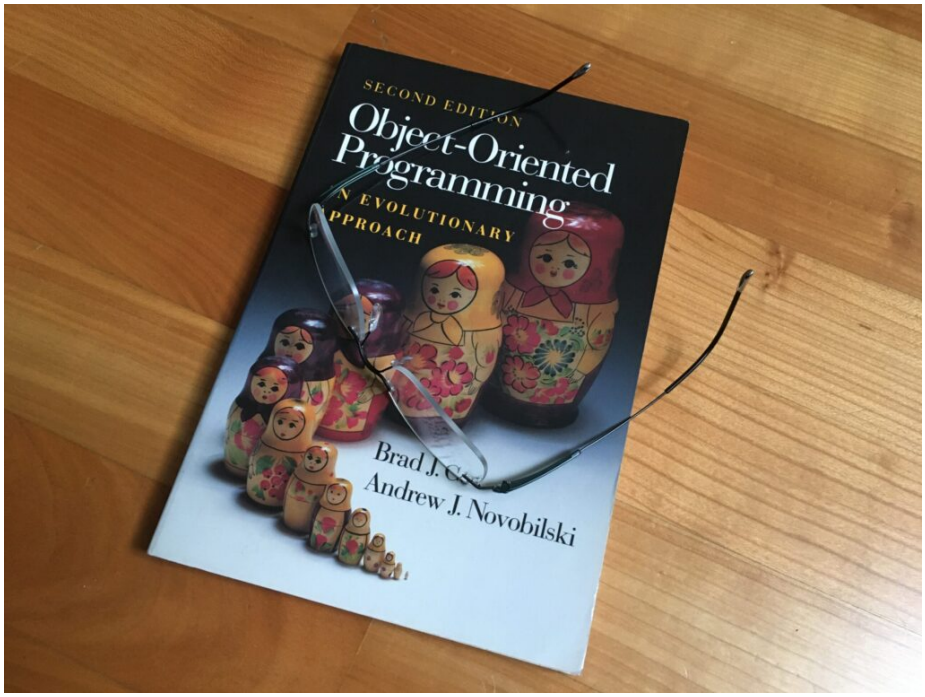
[10] https://unsplash.com/@dsmacinnes?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[11] https://unsplash.com/s/photos/writing?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Brad Cox

Graham Lee

December 2nd, 2019



Ever since Aristotle, philosophers have struggled with the concept of existence. Why do we observe the things that we see in the world? How did they get there? Is there a purpose? Will they exist forever? Are we made of the same things as everything else, or do we have an animus or soul that other objects lack?

In 1714, Gottfried Leibniz published *La Monadologie*[1], in which he proposed that at the fundamental level, objects are made of indivisible entities called Monads. You can't see inside a monad, it has an internal state that controls its actions and passions. Monads can be created or destroyed atomically, but cannot be partially built or partially annihilated because they are a "simple substance without parts".

This doesn't represent the pinnacle of metaphysical exploration. In fact, through the 20th century Martin Heidegger completely upended the whole field. In his 1927 introduction to *Being and Time*[2], he puts forward that you can't ask the question "what is Being?" because the question itself presupposes an understanding of the word "is", in other words it relies on an understanding of Being in which our understanding of the being of Being can be situated.

Thus Heidegger initiated what is now called Modern Hermeneutics, in which "facts" are understood not to represent a fundamentally true world outside our experience, but a working interpretation informed by our experiences, our perspectives, and our histories.

[1] https://en.wikipedia.org/wiki/Monadology
[2] https://en.wikipedia.org/wiki/Being_and_Time

Computing, of course, spent a long time without questioning its relationship to reality. Computers were seen as domains of pure logic, in which either procedures or routines of mechanical calculation instructions were carried out (the Turing school) or functions mathematically transformed data (the Church school). These two paradigms were both silent on the meaning or interpretation of the data being acted upon, which was left as an exercise to the reader.

The concept of being, in a form that logicians would recognise as ontology but Heidegger would call merely ontics, entered computing at first slowly, and then all at once. (Ontology, Heidegger says, is the investigation of Being, while Ontics is merely the investigation of things that are in a system in which Being is accepted.) Simula broke down the barrier between the things outside the computer and the procedures written in the computer by introducing software classes. Now programmers could say "this piece of the computer represents a star", or an animal cell, or an employee, and they could say "all [simulated] stars have this information about themselves" and "all [simulated] stars can perform these procedures".

Researchers at Xerox PARC developed these ideas in the context of Alan Kay's Dynabook[3] concept and the Smalltalk programming environment, finally publishing information about their "object-oriented" programming system in the August 1981 issue of Byte magazine[4]. The objects in Smalltalk clearly reflect a Leibnizian Monadic simulation of the world. Objects (monads) can be created or destroyed, but either exist or they do not. They encapsulate private data, which is inaccessible from outside but can be willed into action including changing their internal state by the receipt of messages. Crucially, an object itself decides what to do when it receives a message, a level of indirection and isolation not found when invoking a named procedure.

Taking full advantage of this way of structuring software meant thinking entirely differently about how software is designed and built. Daniel Robson, one of the contributors to the Byte issue on Smalltalk, acknowledged that it would be easier to come to objects fresh than to shift paradigm from a procedural view of software.

> ...the basic idea about how to create a software system in an object-oriented fashion comes more naturally to those without a preconception about the nature of software systems.

Object-oriented programming removes a conceptual barrier between the things we're building software for and the software being built. All of the people, processes, organisations, artifacts and natural resources in the real world can have analogous software avatars in the form of objects, interacting with each other by sending messages within the software system. A small collection of programmers, authors and educators in the 1980s saw that this meant not only changing how you write software, but how you think both about the software and the context in which you are creating that software.

Brad Cox was one of those people. On the surface, Objective-C looks like a tool for designing objects that internally execute C procedures, and indeed that is what it does. But Objective-C was born of a desire to do for software what Intel had done for hardware, and commoditise algorithms and data structures in a component model analogous to the integrated circuit.

In *Object-Oriented Programming: an Evolutionary Approach*[5], Cox laid out this vision for "Software-ICs". It is not fair to make the usual comparison that this book is to Objective-C

[3] https://en.wikipedia.org/wiki/Dynabook
[4] https://archive.org/details/byte-magazine-1981-08
[5] https://openlibrary.org/books/OL1866403M/Object-oriented_programming

as K&R is to C, or Stroustrup's book is to C++. K&R is a user's manual for a tool, and the C++ book is a guide to making effective use of a complex design system. OOP:aEA is nothing short of a manifesto for a completely different way to funding, staffing and delivering software products.

The Kernighan and Ritchie book[6] opens with the "Hello, World" example. *The C++ Programming Language*[7] follows an annotated table of contents with "The purpose of a programming language is to help express ideas in code." Cox, on the other hand, opens with the story of Eli Whitney and the industrial revolution.

For Cox, the industrial revolution is not primarily about machinery and the harnessing of steam and coal power. It's about the replacement of artisinal, cottege manufacture with scaled-up industrial processes that depend on well-specified interfaces between standardised, interchangeable parts. Where previously gunsmiths made one-off rifles with an end-to-end process, Whitney argued for rifles to be assembled from standard components. Then, if a bolt fails in the field, you just need to replace the bolt, not the rifle.

Cox takes an object-oriented look at *software production*, in much the same way that Ivar Jacobson took an object-oriented look at business processes. He wanted programmers to design objects, and publish data sheets that described how the objects worked. Integrators would browse these data sheets, and buy objects that they could connect together to build higher-level assemblies that solved their problems.

Ten years later, Cox had to accept that while the company he had co-founded to promote Objective-C had found some financial success, it was far from the Intel of software. To find out why, we pick up the story of another of his books, *Superdistribution: Objects as Property on the Electronic Frontier*[8].

Again, the story of Eli Whitney and the discussion of a "software industrial revolution" comes up. But now we get to the difficulty with scaling component sales in software: we don't sell software by the unit, for the most part. Most of the cost is borne in the initial construction, and subsequent copies are free. We can't just buy a couple of Window objects to kick the tyres before scaling up our production of graphical applications: we either license infinite Windows (and probably the rest of the GUI toolkit) or none.

Cox advocated for an object-oriented economy. When I use my application, my computer registers that use and makes a micropayment to the application author. However, the application also makes use of the GUI toolkit objects, and a portion of the payment is transferred from the app author to the toolkit author. If the toolkit uses some foundational data structures, then another payment is made.

Now development is cheap, because a developer pays less when they only use a couple of objects on their own computer. But when they distribute the application, and more users start using it, their revenue and the share they pass "upstream" both increase with the scale of the app's use.

As described in *Superdistribution*, object-oriented economics required custom hardware to securely track object usage and make the correct micropayments. A small trial in Japan was not followed up. But Cox was writing in the pre-blockchain times, before dApps written in Solidity made use of micropayments in the form of smart contract evaluations on the

[6] https://openlibrary.org/works/OL4617640W/The_C_Programming_Language
[7] https://openlibrary.org/works/OL53184W/The_C_programming_language
[8] https://openlibrary.org/works/OL2985873W/Superdistribution

Ethereum network. We have the technical capability to implement the ideas of Superdistribution, and we still have the problem of charging a fair amount of creating software components.

The only innovation to succeed at promoting the distribution of software components at scale is the free software license, which doesn't make provision for paying its creators. Superdistribution deserves a modern reading, and the ideas behind a software-industrial revolution still have a lot to inform our field.

Cover photo by Adrian Kosmaczewski.