Issue 004: Programming, Art Or Science?

Adrian Kosmaczewski

January 7th, 2019

Happy 2019 and welcome to the fourth issue of *De Programmatica Ipsum*, dedicated to the subject of *Programming, Art Or Science?* In this edition we have two guest writers:

- Carola Nitz[1] explores the boundaries between science and art[2].
- Roland Leth[3] finds the contact point[4] between both perspectives.
- Graham explores the art on Donald Knuth's classic book[5] "The Art of Computer Programming".
- In this issue's subscriber-only article, Adrian reviews some interesting moments[6] in the "Science vs. Art" debate in literature and programming history.

Enjoy this issue! Please let us know if you have any feedback[7] and get our free newsletter[8] to stay updated about new releases. If you want to support us, subscribe[9] for a month or a year, and let us know if you would like to write with us[10].

Cover photo by Nhu Nguyen[11] on Unsplash[12].

[1] https://deprogrammaticaipsum.com/user/caro/
[2] https://deprogrammaticaipsum.com/the-creativity-of-computing/
[3] https://deprogrammaticaipsum.com/user/rolandleth/
[4] https://deprogrammaticaipsum.com/why-not-both/
[5] https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming/
[6] https://deprogrammaticaipsum.com/a-brief-history-of-programming-artists/
[7] https://deprogrammaticaipsum.com/feedback/
[8] https://deprogrammaticaipsum.com/newsletter/
[9] https://deprogrammaticaipsum.com/subscribe/
[10] https://deprogrammaticaipsum.com/write-with-us/
[11] https://unsplash.com/photos/kr-jmsASg8M?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[12] https://unsplash.com/search/photos/keyboard-art?utm_source=unsplash&utm_medium=referral&utm_c

ontent=creditCopyText

The Creativity Of Computing

Carola Nitz

January 7th, 2019

When we think of Programming we might think of the creative process. The one that enables us to solve complex problems with well designed and engineered algorithms, using advanced math. And at the same time by creating mesmerizing, intuitive interfaces that make everyday problems easy to solve for the entire world.

We all know that, like every article that has ever asked this question, we will conclude here that programming is in some way both artful and scientific. Is there ever any doubt about it? With that being said, in my book the proportion of what is science and what is art has shifted over the years; but let us take a closer look.

Who Does The Programming?

The people who perform tasks that are associated with programming have many names. They are called Software Engineers, Programmers, Developers or even Hackers, by definition or education Computer Scientists! When we learn programming we learn about all aspects that come with it and education starts from the ground up. We need to understand the physics behind flowing or not flowing electricity and how it forms our 0s and 1s. To learn how hardware components like diodes, resistors and transistors work together and how they form the basis to everything we work with by building blocks we use every day. We need to comprehend how to combine those ANDs, ORs and NORs, how they work, and what logic behind them creates the outcome we witness.

Over years we pore over books to create an image of how all of these maps to the programming languages we use today. Then we write our first components and learn how to construct them together. We sit through classes of Math and Algorithms and Linear Algebra, in order

to apply those rules. They are useful to solve routing problems for Mapping Services, or to build physic engines for games to mimic how objects move. We use Geometry and vector math to build interfaces that photographers can use, to crop and rotate their images, or to interpret digital signals from an MRI or CT to show a visual representation of a patient's body.

Our knowledge of geometry lets us display what was once hidden from the naked eye, but that is not all. These components need to communicate with each other and we get all these different architecture models at hand and acquire vast knowledge about various patterns which we later try to apply in different ways.

Scientific Programming

The approach we use while developing is very scientific as well. We have a hypothesis of how things should work and communicate together and based on that, we apply and design our code and components. Along the way we might find obstacles that we did not think of, so we adjust our hypotheses with new parameters and try again. Rinse and repeat until we achieve our goal. We learn about space and time complexity and how they set limitations to what we can build, but none the less we are not seldom astonished what people create with programs and the zeros and ones they are given. Tasks that once took months or years are just some quick display touches away, computations and simulations done quickly.

This all sounds very dry, logical and lonesome, which was further embodied by the stereotypical image that society had of a programmer. Jurassic Park for example pictured us as the nerdy boy with the glasses that obsessed over his computer in his basement and media portrait a programmer as a person with glasses in solitude in front of their screen immersed by darkness, the face only illuminated by the glow of their screen.

The Art Of Creative Coding

But programming has changed over the years! We do not need to understand anymore all the underlying mechanisms to build software. It got more and more abstracted by frameworks so that we can concentrate on implementing functionality. The image of a programmer has evolved as well. These days it is understood that huge parts of our work involve original ideas, creative problem-solving skills, and even greater communication skills. Without them, none of the above would be possible. We are often faced with hard to overcome challenges, and team up with colleagues to come up with brilliant solutions to these.

We feel deep satisfaction when we finally find an easier, elegant way to solve problems, as part of a great architecture that is simple enough to be understood by the ones that come after us. After all it is a direct reflection of the end product, and do not we all marvel at the code of others that is readable, comprehensible and straightforward? *Make something complex appear easy is a trade that takes years to develop.*

A lot of time is spent learning new technics, architectures, and styles and experimenting to find what works best given a certain problem. We see it as a craft that we refine with time, seek out teachers and masters, go through books, attend workshops and conferences to add yet another tool to our toolset to better our work. We explore different languages and environments to find the place where we feel comfortable. And all the while we never stop learning as everything around us keeps evolving and changing but though core concepts stay the same.

Developing A Programming Style

Over the years we evolve and look back at our earlier works, often with criticism or shame. Au contraire to what we feel, it is a good sign, an indication that we have grown. With more time we start developing a certain programming style, that makes us to the programmers we are today, formed by our journey. What you started out with often defines how you code. Was it a functional language or object-oriented, do you know how to work with pointers and did you have to manage memory yourself?

All of that defines your next approach with a new language. Soon enough we have a coding style that we apply and a way how we arrange variables, functions and where we put our brackets. You can also see our personal signature in the way our components communicate with each other, the way we handle errors and structure modules. We all know that one colleague who uses templates or protocols everywhere or who never writes more than 80 characters in one line. We do not even need to open the commit log to know who has written a certain piece. It is all there green on black. A simple look at the code structure is enough to know who built this. The significance of ones own code has even gotten so far that a project was developed by stylometry researchers to identify the anonymous authors of code[1].

> There is a very strong stylistic fingerprint that remains when things are based on learning on an individual basis.

Everyone has their own unique style, no initials needed on the masterpieces you create. Very much influenced by the ones we worked with just like painters who teach their students, seniors brush off on their juniors.

The March Of Progress

Not only the underlying technical components have changed and evolved. The devices we work with have also become so powerful that we are able to build very computing intensive animations and visual elements to give our programs a unique touch. We are not bound anymore to standard elements, and can create interactions with our programs that will influence generations and pop culture. The interface elements and components created for popular programs shape our communication. For example sentences like “I would give it a like” or “I’d swipe right” have made its way into our everyday vocabulary to express approval. Both of these go back to parts of the widely used social platforms Facebook and Tinder.

Who Are Your Heroes?

Just like painters and other artists we programmers create master pieces in the hopes to one day proudly present and share them with the world. It even goes so far that some of our pieces are presented on giant stages, as the main act of a big presentation in artful orchestration. Received by applauding crowds that flew many miles and purchased expensive tickets and accompanied by other artists like U2. We have our idols that inspire us like Bill Gates, John Carmack and Steve Jobs who have created computer programs that have changed the way we live and work today.

And creating something that changes the world and makes a lasting impact is certainly what many programmers aspire but in order to influence the masses it is not enough to solve a problem with a piece of software. We need so much more than just math and algorithms turned into code. Our creations need to be intuitive and simple and delight but yet powerful.

[1] https://www.wired.com/story/machine-learning-identify-anonymous-code/

Achieving this needs a lot of work, many iterations, failures, set backs and from that learnings to make it all the way from an idea to a clunky prototype up to a refined piece of software. Often this software is never done, and requires a certain amount of empathy since it needs to evolve. We need to understand our users if we want to stay relevant, not only for our Applications but also the users of the framework we build.

What Is Programming?

So is programming Science or Art? It is a life long question ever since programming became popular and I do not dare to put a label on it since it depends very heavily on what you use it for.

Photo by Federica Galli[2] on Unsplash[3].

[2]https://unsplash.com/photos/Nl-k9CcqY4Q?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[3]https://unsplash.com/search/photos/camera?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Why Not Both?

Roland Leth

January 7th, 2019

There is a question that has been around forever, and that we love tossing around now and then: is programming science or art? Some are adamant about the fact that it is related to computers, and it is very technical, so it is surely science. Are they (completely) right?

I will start with the obvious science-y part of programming. At first, we learned how computers work. We learned how programming languages translate into computer language. We learned how programming languages are composed, how they are structured, how they *work*.

From Science To Art

We learned that at some point there were not any! We "talked" to the computer through physical punched cards[1]. That was still a form of programming.

Then we started writing our own software, and ever since, we struggle to squeeze as much performance out of it as possible. We learn to apply the best algorithms, and in case of the UI, to use those algorithms in such a way that UI performance is not hindered. We strive to deliver the fastest network responses, the fastest file loading, or the fastest frame rates.

We learned how *hardware* works, if we chose such a field, so we can correctly program whatever device we are working with. We need to know what the device can or can not do, at the

[1] https://en.m.wikipedia.org/wiki/Punched_card

very least, so we know *if* what we are trying to accomplish is even possible. This might be thought of as software still, to a certain extent.

Even if we did not choose such a field, we might still learn how certain hardware or parts of it work, like networking, displays or storage:

- We struggle to keep our binary sizes small.
- We think and care about the efficient communication between our software and the device it runs on, for example when working with local storage or aiming for a high frame rate.
- We also think about the communication between our software and *other* devices, and everything in between, for example when working with cloud storage or Bluetooth devices.

We learn and try to use the newest and best features a language and/or framework has to offer. Newest sometimes means improved, optimized, faster, more efficient, so it makes sense to do so.

We learn different programming languages, as to expand our knowledge, mindset and field of view; to learn how "others do things"; to learn new paradigms. It helps us grow.

We hunt for bugs and sometimes we turn our minds inside out to solve them. We have to rewrite entire parts of our software because the initial science behind it was wrong and there is no way around it; we have to start from scratch.

We strive to follow the best practices of the language we are working with. That is why they are "best practices", because they have stood the test of time and the majority agrees over them.

We even learn about the science of human nature and behavior. We all know quite a few examples of shady practices in emotion manipulation, but I hope all of you use it for a good cause, as in creating a good user experience with buttons that are obvious, text that is readable, controls that are easy to use and easy-to-follow flows.

Transition

Have you noticed when we started moving from science to art? It happened a few paragraphs above, when we mentioned *learning* new programming languages. The computer does not care what paradigm we are using; it does not care if we follow the best practices; it will relentlessly execute *exactly* what we tell it to. So we better write in such a way that *we* understand what it does, and easily, otherwise we are the ones in trouble.

Writing clean code. Building a correct architecture. Laying out a correct project structure. Clean names. Paradigms. Architecture. *All of this is part of the artistic side of programming.* Part of the "delivering an idea, a concept, a message to another human" side of programming.

Writing clean and well structured code is of no concern to the computer, there is no science behind it. But, just like a piece of art, it delivers a message in a clean way; it is easier to look at, more pleasurable to look at and easier to understand; it grows on you.

Laying out a correct project structure is of no interest to the computer, it serves ourselves to more easily find what we need; to better understand how pieces fit together and how the data flows. Just like a piece of art puts into perspective and emphasizes the most important parts, but also guides your eyes from one area to another.

Clean names and paradigms affect the computer in no way. It does not care if we name a method `perform` or `performFadeAnimation`. But it will help us to better understand what the instructions are. Just like a work of art that has its elements properly defined, balanced and obvious.

Our Audience

Ultimately, we do not write code for the computer, we write code for humans. The computer will not care how we lay out our code, structure, or name things; you could have one line of code that contains your whole program, the computer would not care, as long as the science part would be correct.

But may the Universe have mercy on you when you will want to read what you wrote there. Just like an ugly painting, you would rather throw it away than look at it, not to mention enjoy or make use of it. You have seen it. You have been handed it. You have *written* it. We all did. And we all cried because of it; we all wished it was not so.

On the other side, think of the times you stumbled upon good code. Properly written. Properly structured. Easy to understand. It felt inspiring. It felt like looking at a work of art. And it was!

Be An Artist

Do not just write code for the sake of telling the computer what to do. Do not just care about the science-y part. Strive to write understandable code. Beautiful code. Correct code. Because at the very least, you will be the one looking at it in the future, if not others as well.

Just like a painter learns how colors blend, how shapes emphasize or affect emotions, how different brushes work, what kind of paints to use for what purpose, or what kind of canvas, so we have to learn the science behind computers, programming and devices we will work with. To know what tool to use for what purpose, what language for what project and what architecture for it, to understand how different devices work so that we can interconnect them, or what particularities the device we are building for has.

But just like a painter strives to deliver a clear message, to properly structure a painting, to emphasize certain aspects of their creation and to guide your eyes easily throughout the story, we have to do the same. We need to send a clear message, to build an easy-to-follow story and to emphasize the important bits. We need to know what canvas to use, what paints work properly on it, what colors and shapes to use, what brushes to use and for what purpose.

Use science to understand how programming works, to build with it. Use art to decorate it, to easily send the proper message to the next person after you, even if that person is you.

Cover photo by Clem Onojeghuo[2] on Unsplash[3].

[2]https://unsplash.com/photos/xJXxMR5PXoY?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[3]https://unsplash.com/search/photos/artists-code?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Art Of "The Art Of Computer Programming"

Graham Lee

January 7th, 2019

A quote from the cover of Volume One of Professor Donald Knuth's Magnum Opus, *The Art of Computer Programming*[1] (3rd edition):

[1] https://www-cs-faculty.stanford.edu/~knuth/taocp.html

> If you think you're a really good programmer... read (Knuth's) Art of Computer Programming... You should definitely send me a résumé if you can read the whole thing.

"Me" here refers to the author of the quote, Bill Gates. This book (TAOCP hereafter) is a defining work in our field. The reason Gates would like your résumé is that the reason most of us bought our copies in the 1990s was twofold:

1. We wanted to look like the sort of people who *had read* TAOCP; and
2. They were handy for raising heavy Cathode Ray Tube monitors on our desks.

Gotta Catch 'em All

Bill Gates has never received a résumé from anyone who read all of TAOCP. There's an urban legend (now sadly debunked[2]) that Steve Jobs told Knuth he had "read all of his books", to which Knuth replied "you are full of crap".

While that tale is not true, nobody including Steve Jobs can say that they have read all of TAOCP. The reason is simple: it isn't finished yet. I foolishly bought my copy of volumes 1-4A while abroad at Apple's WorldWide Developer Conference, and had to travel back to the UK with it in my case. There are six fascicles available that comprise volumes 4A-4B. The predicted scope includes more sub-volumes 4, then volumes 5-7.

As all of volumes 4B-7 are somewhere between partially and completely incomplete, nobody has read all of TAOCP, Donald Ervin Knuth included. Therefore Gates has never received such a résumé.

The Art Of Yak Shaving

Some of the reason it has taken so long to produce the seven volumes is that Knuth is the master, maybe even the patron saint[3], of yak shaving.

In the Jargon File, Yak Shaving[4] is a task you complete so that you can complete a task so that...

While reviewing an updated edition of TAOCP, Knuth saw that the pages were decidedly lower quality than the earlier edition. His printers had changed from a hot metal typesetting process to a computerised process to lay out text on the page, and it did not produce good results. Knuth felt that TAOCP needed to *look* and *feel* like a higher quality book.

He stopped working directly on TAOCP to create TeX[5], a text layout system based on virtual boxes and glue. To make families of related-looking fonts, so that italic text rendered in TeX looks similar to fixed-width typewriter text, roman text and other styles, he created METAFONT[6], a programming language for generating fonts from descriptions.

Meanwhile, he wanted to be able to describe his programs both for the machine to run, and for people to understand. So Knuth made Web, a Literate Programming[7] tool that could "tangle" source into a Pascal program for the computer and "weave" it into a TeX document for a reader.

[2] http://www.catonmat.net/blog/don-knuth-steve-jobs/
[3] https://yakshav.es/the-patron-saint-of-yakshaves/
[4] http://catb.org/jargon/html/Y/yak-shaving.html
[5] https://www.tug.org
[6] http://wiki.c2.com/?MetaFont
[7] http://www.literateprogramming.com

So Web allowed him to make TeX and METAFONT (and the five-volume Computers and Typesetting[8] collection). TeX and METAFONT allowed Knuth to get back to TAOCP.

Sir MMIX-a-Lot

Meanwhile, the world of computing had not stood still. The original volumes 1-4A described algorithms implemented in an assembly language called MIXAL, targeting the hypothetical MIX computer. MIX contains features no longer found in popular computer architectures. Memory can be accessed as six-bit bytes, either in binary or decimal. Bytes are grouped into five-byte words, which each has an external sign bit. Dedicated device I/O instructions provide access to paper and magnetic tapes, a card punch, a card reader, and similar technology.

To reflect more recent advances in computing hardware, Knuth produced the MMIX architecture in collaboration with designers of the MIPS (John L. Hennessy) and Alpha (Richard L. Sites) CPUs. MMIX is a 64-bit RISC architecture with lots of general-purpose registers, and IEEE 754 floating-point arithmetic. MMIX was published in Volume 1 Fascicle 1 of TAOCP in 2005. Most of the rest of the book still uses MIXAL source code. But here we encounter another hairy yak: will Knuth press ahead with writing volumes 4B-7, or will he re-implement the volumes 1-4A algorithms in MMIXAL?

Further yaks: should Knuth move on to volume 4B or 5, he will find things that *should* have been in volumes 1-3 but did not exist when they were written. He is one man, trying to capture developments (the state of The Art, even) of a whole industry of commercial practitioners and academics. Like Lewis Carroll's red queen, he must run in order to stand still.

So Is It Art?

TAOCP is definitely a work of art. Knuth has taken a great deal of care over the content, its preparation and its presentation. He continues to do so. The descriptions of algorithms in the book are clear: I have re-implemented a few of the algorithms where application performance required it. I found the discussions easy to understand and follow, so that rather than translate the MIXAL into C or some other language I was able to write my own implementation that followed the text.

There are many books that have been more useful in my career: NeXT's developer documentation, Larry Wall's "camel book", and Aaron Hillegass's *Cocoa Programming for Mac OS X* have all taught me more *immediately useful* knowledge on making software. But TAOCP occupies a special place as a book that is worth reading *for the pleasure of reading it*, a rare creature in the realm of software documentation.

My shelf of such books would be fairly small. Alongside TAOCP would be found *The Structure and Interpretation of Computer Programs*, *Numerical Recipes*, *The Art of the Metaobject Protocol* and *Object-Oriented Software Construction*. Maybe I could identify a handful of others. Perhaps others (including you?[9]) will suggest a few more. I do not believe that I will be stretching Vitsoe's logistics[10] to find shelving for all of the works of art in computing writing.

[8] https://www-cs-faculty.stanford.edu/~knuth/abcde.html
[9] https://deprogrammaticaipsum.com/feedback/
[10] https://www.vitsoe.com/gb/606

And Is It Science?

Many would say that TAOCP is indeed a book on science. *American Scientist* describes it as one of the best monographs on physical science. Knuth has been elected to the National Academy of Sciences and to equivalent bodies in France, the UK, and possibly elsewhere.

Is it sufficient to say "people call this book a science monograph, and say the author is a scientist, therefore this book is about science"? Is computer programming a scientific discipline? For that matter, is computer science a science?

The Origins Of Computer Science

The phrase "computer science" has been used by academics since at least the 1960s (Cambridge University's 1953 program was in "Numerical Analysis and Automatic Computing"), and was popularised by the Association of Computing Machinery in its curriculum recommendation. In effect this was a political move. "Computer science" can be seen to lend a gravity to the discipline that more abstract words like "Informatics" did not imply.

Uncharitable readers may note that "political science" and "management science" also seem to attach themselves to the intellectual heft of science. However, in the case of computer science, there is some additional support. Computer science is not so much the science of computers but the science of information. Computers themselves are the products of semiconductor physics and electronic engineering.

But computers are not about band gaps or voltage levels. When we use a computer, we do not use it so that we can induce free electrons in lumps of silicon. Computers are tools for information processing. Computers thus are the experimental labs in which information science is performed. Leibniz's binary logic and Boole's binary algebra give us language and philosophy with which to consider *any* problem. Computers are tools to let us express and manipulate those problems.

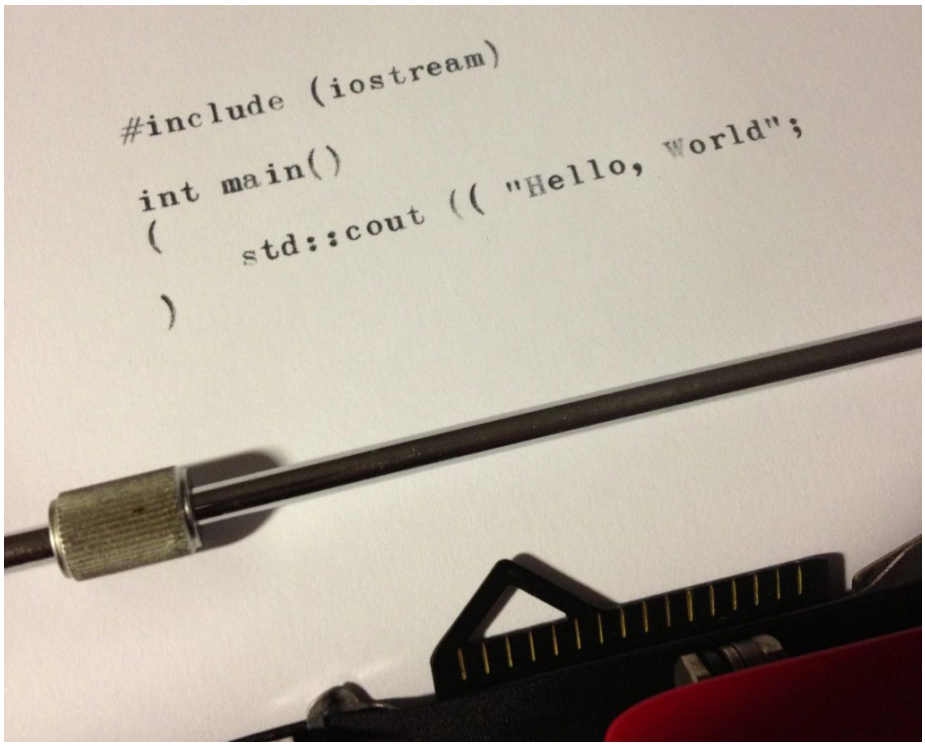
TAOCP is a book about algorithms in binary logic. It is a book about the applications and characteristics of the philosophy and logic to the real-world situations and problems we are modelling. It is very much about science.

Image credit: author's own.

A Brief History Of Programming Artists

Adrian Kosmaczewski

January 7th, 2019



The literature of software development contains many references to the "Art vs Science" dichotomy. This article presents a rather short and recessarily incomplete overview of projects, books and papers referring to the subject written in the past 50 years.

Programming As Artisanal Work

A friend of mine,[1] musician turned software developer, once told me that software developers are not so much artists than artisans. His reasoning went as follows: as beautiful as code can be – regardless of your definition of the world "beautiful," – software is always somewhat defined by its *utility*. On the other hand, art exists per se, without any kind of intrinsic utility rather than its mere existence, and the sensations it generates to its public.

A piece of software, following this reasoning, must be useful, and hence is not actually art.

But it might as well be classified as *artisanal,* in the sense that it conveys utility and beauty, just like pottery, knitting garments, or traditional musical instruments made by a *luthier*.

In the third edition of "Free Software, Free Society"[2] by Richard Stallman, he appears to agree to this definition, from chapter 17, "Words to Avoid (or Use with Care)":

[1] http://hernun.com.ar/

[2] https://www.gnu.org/philosophy/fsfs/rms-essays.pdf

> To speak of "consuming" music, fiction, or any other artistic works is to treat them as products rather than as art. If you don't want to spread that attitude, you would do well to reject using the term "consume" for them. We recommend saying that someone "experiences" an artistic work or a work stating a point of view, and that someone "uses" a practical work.

I remember back in the 90s, my team was using components made by a company named "Software Artisans,"[3] relatively well-known in the Microsoft galaxy as a producer of prebuilt COM components to be used in Windows and ASP applications.

The word "artisan" conveys a lot of positive values, indeed, including attention to detail, efficiency, quality and support. Countless papers and books touch, directly or indirectly, the world of software craftmanship.

Craftmanship. Even this word suggests cozy feelings and good vibrations.

Programming Languages As An Artistic Medium

Just like thousands of other developers, I discovered the Ruby programming language[4] back in 2005, at a time when Ruby on Rails[5] was becoming a storm that was about to change the face of web development forever.

Yukihiro "Matz" Matsumoto[6] created Ruby to become a tool to make better, more beautiful things, making developers happier and more productive. (Maybe he wanted them to become better artisans?) He mentioned this intention several times in interviews, books and blog posts. His work, started in the early nineties (roughly at the same time as Python,) was virtually unknown to western developers. The "Programming Ruby"[7] (also known as the "Pickaxe" book) by the Pragmatic Programmers was the first (and for a while the only) guide to the language published in English.

Inspired by this philosophy, a developer only known by the nickname of "_why the lucky stiff"[8] or simply "_why," wrote The Poignant Guide to Ruby,[9] which might as well qualify of the weirdest, funniest, most radically different, programming book of all time. In a sequence of vignettes, mixing tweaked pictures, drawings, and code snippets, _why introduces Ruby, its syntax, and its philosophy.

Unfortunately, in 2009, _why decided to delete all of his online presence. Thankfully there are some online collections[10] of his work.

Programming As A Self-Fulfilling Prophecy

Tom Murphy[11] wrote a "strange paper"[12] for the 2017 SIGBOVIK conference, explaining the structure of a small, experimental C99 compiler called "ABC." The interesting part of this work is that his paper *is* the actual compiler. Once saved, a user can execute the contents

[3] http://www.softartisans.com/
[4] https://www.ruby-lang.org/en/
[5] https://rubyonrails.org/
[6] https://en.wikipedia.org/wiki/Yukihiro_Matsumoto
[7] https://pragprog.com/book/ruby/programming-ruby
[8] https://en.wikipedia.org/wiki/Why_the_lucky_stiff
[9] https://poignant.guide/
[10] https://whymirror.github.io/
[11] https://twitter.com/tom7
[12] http://tom7.org/abc/

of the paper (which is also 100% printable) from a Windows command line, and actually use it to generate EXE programs.

> Executing this paper in DOS, with an AdLib-compatible sound card (such as the Sound Blaster) configured at 0x388, will play some music. The music to play is specified on the command line, using a subset of a standard text-based music format called ABC [ABC'05]. For example, PAPER.EXE C4C4G4G4A4A4G8G8F4F4E4E4D4D4C8 will play a segment of the "Now I know my ABC's" song and then exit.

The paper also contains its own source code, and the author even tried to make it self-printable. During the preparation of this article I installed FreeDOS[13] in a VirtualBox machine in my laptop, copied the paper and executed it, and to my amazement it worked perfectly well, as intended.

Something tells me that Alan Turing would have loved to read this paper.

Programming As An Art School

These days you can become a poet programmer. The School for Poetic Computing[14] defines itself as:

> ...an artist run school in New York that was founded in 2013. A small group of students and faculty work closely to explore the intersections of code, design, hardware and theory — focusing especially on artistic intervention. It's a hybrid of a school, residency and research group.

One of the SFPC teachers, Taeyoon Choi,[15] wrote a book about the subject, called "Poetic Computation: Reader"[16] exploring the boundaries between both worlds.

> I'd like to begin the class by asking "What is poetic computation?" First, there is the poetics of code, which refers to code as a form of poetry. There is something poetic about code itself, the way that syntax works, the way that repetitions work, and the way that instruction becomes execution through abstraction. There is also what I call the poetic effect of code, which is an aesthetic experience realized through code. In other words, when the mechanics of words are in the right place, the language transcends its constraints and rules, and in turn, creates this poetic effect whereby thought is transformed into experience.

If you are in the west coast, you might be more interested in participating in Dynamicland[17]:

> We are a non-profit long-term research group in the spirit of Doug Engelbart and Xerox PARC, inventing a new computational medium where people work together with real objects in the real world, not alone with virtual objects on screens. We are building a community workspace in the heart of Oakland, CA. The entire building is the computer.

[13] http://www.freedos.org/
[14] http://sfpc.io/
[15] https://twitter.com/tchoi8
[16] http://poeticcomputation.info/
[17] https://dynamicland.org/

Programming As The Antithesis Of Art

In "Mindfire: Big Ideas for Curious Minds,"[18] Scott Berkun makes a point about the characteristic that defines what an artist is, and which might work as a guide for a few of us in the field.

> But if you work for clients/bosses in the making of things that you yourself would not consider art, or are beneath your own standard, or that you blame others you work with for ruining, you are not an artist. You are an employee. You are being paid to give someone else authority over your creative decisions. This can involve inspiration, effort, sacrifice, passion, brilliance, and many other noble things, but it's not the same as being an artist.

Programming As Intuition

A recent paper from Harvard University, "The Periodic Table Of Data Structures"[19] argues that art is driven by inspiration:

> Art vs. science. Just because a specific design is a combination of existing design concepts, it does not mean that it is easy to be conceived manually. In other words, when the number of options and possible design combinations is so big, it is hard even for experts to "see" the optimal solutions even when these solutions consist of existing concepts only. This is why many argue that research on algorithms and data structures is a form of art, i.e., driven by inspiration. Our goal is to accelerate this process and bring more structure to it so that inspiration is complemented by intuitive and interactive tools.

Programming Artistry As A Problem

In the classic 1968 "Software Engineering: Report of a Conference Sponsored by the NATO Science Committee"[20] paper, the relationship of programming with art is seen as somewhat problematic, as something that should be fixed for the profession of programming to become a branch of engineering. It remains to be seen whether there has been any progress in the 50 years that passed since the publication of this paper.

> Several participants were concerned with software engineering management and methodology, as exemplified by the following remarks. d'Agapeyeff: (from Reducing the cost of software) "Programming is still too much of an artistic endeavour. We need a more substantial basis to be taught and monitored in practice on the: (i) structure of programs and the flow of their execution; (ii) shaping of modules and an environment for their testing; (iii) simulation of run time conditions."

Programming As Joy

The conclusion of this article is that... well, there is no conclusion. Art or Not Art, it does not matter; programming is something we do. In the humble opinion of this author, **the joy of programming is what makes it an art;** we become artists the moment we find happinness while making things happen in a computer.

[18] http://scottberkun.com/the-books/mindfire-big-ideas-for-curious-minds/
[19] https://stratos.seas.harvard.edu/files/stratos/files/periodictabledatastructures.pdf
[20] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

And so it happens that this joy is personal, and most often than not, very hard to share with others. We all feel sometimes like a Mark Watney[21] celebrating in Mars, all alone, that something worked. We know that feeling.

Maybe *that* feeling is the feeling of an artist. Or maybe not.

Maybe it just does not matter. Maybe, just maybe, things just *are*.

Coda

I will leave these references here for you to explore, some of which are well known, some might not; they all touch the subject of artistry in programming in some way or another. Enjoy!

- "The Mythical Man-Month"[22] by Frederick Brooks.
- "The Psychology of Computer Programming"[23] by Gerald M. Weinberg.
- "Hackers and Painters"[24] by Paul Graham.
- "The Art of Computer Programming"[25] by Donald Knuth.
- "Object-Oriented Software Construction"[26] by Bertrand Meyer – for example, on page 878, when he mentions Barry Boehm.
- "A View of 20th and 21st Century Software Engineering"[27], a paper by Barry Boehm from 2007.
- "Dealers of Lightning"[28] by Michael Hiltzik.
- "Revolution in the Valley"[29] by Andy Hertzfeld.
- "The Tao of Programming"[30]
- "Masterminds of Programming"[31] – for example on page 305, where Anders Hejlsberg talks about C#.
- "Design for Hackers"[32] by David Kadavy.
- "The 'Future Book' Is Here, but It's Not What We Expected"[33], article on Wired from December 2018 by Craig Mod.

Cover photo by the author, made with an Olivetti Lettera 22 typewriter.

[21] https://en.wikipedia.org/wiki/The_Martian_(film)
[22] https://en.wikipedia.org/wiki/The_Mythical_Man-Month
[23] https://leanpub.com/thepsychologyofcomputerprogramming
[24] http://www.paulgraham.com/hackpaint.html
[25] https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming
[26] https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction
[27] http://csse.usc.edu/TECHRPTS/2006/usccsse2006-626/usccsse2006-626.pdf
[28] https://www.amazon.com/Dealers-Lightning-Xerox-PARC-Computer/dp/0887309895
[29] https://en.wikipedia.org/wiki/Revolution_in_the_Valley
[30] http://canonical.org/~kragen/tao-of-programming.html
[31] http://shop.oreilly.com/product/9780596515171.do
[32] https://designforhackers.com/
[33] https://www.wired.com/story/future-book-is-here-but-not-what-we-expected/