De Programmatica Ipsum
Volume One
October 2018 to March 2019

Adrian Kosmaczewski & Graham Lee
with
Ioana Porcarasu, Anastasiia Vixentael, Roland Leth, Carola Nitz,
Agis Tsaraboulidis, Susanna Riccardi, Julia Cacciapuoti, & Sheree Atcheson

Foreword by Graham Lee

Adrian and I started *De Programmatica Ipsum* because it didn't exist. We knew about places to get news about the latest programming tools and frameworks. We knew about some amazing bloggers who had really thoughtful content on their favourite—and maybe least favourite—topics. But what's the thing in the middle? Where do we go to get interesting analysis and deep thoughts on varied subjects, to hear from authors we aren't already following, to go beyond "this week in Javascript" and get into the real meaty topics?

We didn't know the answer, so we created one. It was slow to get off the ground; we delayed the launch of the first article so that we could synchronise its availability with some advertising campaigns we devised, and even then we didn't manage to get a guest contributor in time. But issue 1, "Hype", nonetheless was released, and *De Programmatica Ipsum* entered the world.

The magazine hit its stride with issue 2, "Quality". We are proud to pay our contributors real, actual money for their valuable insights, and my PayPal account's virtual cash register rang to the sound of our first guest article. Ioana Porcarasu is an excellent, skilful head of Quality Assurance who had not had the opportunity to publish a professional article before we approached her. This truly is *De Programmatica Ipsum*'s mission: to find the people that software engineers *should* be hearing from, and to ensure that you *do*.

By the time we got to issue 6, we had the process pretty well down. We were able to celebrate diversity and inclusivity in our issue on the subject by shutting up: other than Adrian's interview questions for Sheree Atcheson, every word in that issue was supplied by an external contributor.

The volume you hold in your hands, or on your screen, represents the first six months of this project. All of the articles—freely available on the web and subscriber-only, by myself, Adrian and our guests—are collected together in this compendium. I'll be honest. We don't have as many subscribers as we would like. That means both that it's harder to see a sustainable future for the magazine, and that we are letting our contributors down by not getting them the audience they deserve.

I hope that reading what we've done so far enthuses you to subscribe and support this project. I truly believe that we are bringing something you won't get anywhere else: in-depth analysis on a variety of topics concerning software engineering and software engineers, borne from extensive experience.

This project would not be what it is without Adrian's expertise, commitment, and ability to get writers to meet deadlines; our many guest contributors and their incisive articles; Unsplash and its community of excellent photographers; Igino Marini and his digitisation of the Fell types; and you, our readers and supporters. Thank you to everyone.

Foreword by Adrian Kosmaczewski

It all started on a chat one Saturday morning.

Graham and I were ranting, as is usual for grown-up software developers, about how much we missed magazines like *Dr. Dobbs*.

I was a subscriber of that legendary title, for years, and I loved the cross-platform, cross-language, cross-everything approach it had. I learnt a lot from Dr. Dobbs. Every month there would be news about some new algorithm, database, programming language or IDE that one could try and learn from. There was history, biographies, opinion, news, and an unhealthy dose of tongue-in-cheek geek humour, ages before Twitter, Github, or Slack.

So was born the idea to create an online magazine. One that would be different in some key aspects.

We wanted to talk about the human side of software engineering. Not only about the frameworks, but about the humans who wrote them; about the humans who chose them for their projects; about their reasons and their problems.

We wanted this magazine to have a strong opinion, but not only ours; we wanted others to collaborate. And we took the bold decision to pay actual cash to our guest writers, something that, alas, is not as common as we would like it to be.

In hindsight, I think that the most important aspect of De Programmatica Ipsum was giving a voice to many authors who had never even blogged once; let alone write an article for a magazine. We coached them the best we could, even if unfortunately in some cases (a very small minority thereof, dare I add) we could not succeed.

It is my personal pride to see threads on Twitter starting to unroll, following the publication of some articles. We hope for these conversations to be and to remain respectful, which is gladly still the case.

My deepest hope is to raise *awareness* in other software developers.

Awareness of our status as the creators of this brave new world.

Awareness of the emotions and feelings of our co-workers, users, stakeholders, product owners, project managers, and all those who (paraphrasing Joel Spolsky) deal with software developers by pure chance or ill luck.

Awareness of our condition of workers, entrepreneurs, and makers.

Our guest writers are the true stars of these first six months: my thanks to Ioana Porcarasu, Anastasiia Vixentael, Carola Nitz, Sheree Atcheson, Julia Cacciapuoti, Susanna Riccardi, Roland Leth, and Agis Tsaraboulidis. And to my partner in crime, Graham Lee, who is one of the best writers I have the chance of calling a friend.

Contents

Issue I

Hype

October 2018

What Is Behind The Hype?

Graham Lee

Image: Dmitry Ratushny on Unsplash

Argumentem ad novitatum - appeal to novelty - is the fallacy of saying that some mindset, position, or artifact is good because it is *new*, and newer must mean better. Argumentum ad novitatum is itself named in Latin because of the equivalently fallacious form of argument, argumentum ad antiquitatem; the old ways are inherently better.

If someone on your team comes in effusing about a new UI framework that was just published by a Silicon Valley unicorn to Github or the brand-new Y Combinator funded NoSQL database that the winners of TechCrunch Disrupt used to build their hack, it can be tempting to dismiss the idea of adopting it because newer doesn't necessarily mean better. That we should choose boring technology. But there's probably good intention behind the new tool, even if your colleague isn't getting that over to you.

While *you* may be hearing about this shiny thing because it is new and exciting, that is probably not the only reason for its existence. Try to look past the excitement and empathise with the people who created it. What problem did they have, and what was it about existing solutions that didn't work for them? Were they really making a new tool so that it would be newer than the existing tools, or did it do something the others did not? Now, is that something you can use?

This empathy is often absent, particularly when the community around an idea gets larger and enters the main stream. David Chapman identifies three categories of people in the development of a subculture: geeks, MOPs, and sociopaths. The geeks are the people who love the thing, care deeply about it, understand it in great detail, and love exploring it, creating it, or talking about it.

As the geeks explore and share their interest, they attract a community of MOPs (short of Members Of Public). MOPs are less interested in the minutiae and the philosophy, but still want to be associated with the subculture and want to be *seen* to be associated with it. They adopt its imagery and lexicon, while putting less effort into creation and comprehension of the core parts of the scene. They also provide useful validation (financial or emotional) for the geeks, by legitimising the scene and boosting its numbers.

When the sociopaths turn up, they are like human cuckoos, displacing the geeks from the centre of the scene in order to reap the rewards that come from selling to the MOPs. They are people who are not fanatics about the basis of the scene, and simply see the opportunity to be had in exploiting its existence. They dilute the core principles on which the geeks founded the scene, even to the point where the geeks leave and the subculture is hollowed out.

While Chapman's article was about scenes and subcultures of the late 20th century, like punk music or the fruitarian diet, we can see parallels in some of the big movements of the soft-

ware industry.

Richard Stallman and others initiated the Free Software movement in the 1980s based on a political or ethical notion that the people buying and operating computers should have certain freedoms to use them for any purpose, and to study, share, and improve the software employed on those computers. Two motivating events, according to Stallman, were the acquisition by his department at MIT of a Digital Equipment Corporation computer and a Xerox printer. Both ran proprietary software, and both had room for improvement. But here they were, in the Artificial Intelligence lab at a prestigious university, and neither *could* be improved by the local skilled and motivated programmers because the vendors believed that they owned the software and that the people trying to use their products did not. Stallman came to realise that he disagreed, and that "ownership" of software was anathema. Later, the term "Open Source" was created by Christine Peterson in a group deliberately seeking to take the activities involved in Free Software and make them palatable to businesses, by removing the ideological connotations attached to the Free Software term. They instead associated the idea of sharing source code with Open Systems, an economic movement promoting interoperability between decidedly proprietary software products like OpenStep and OpenVMS. So it was that Netscape Navigator (the browser that was the predecessor to Firefox) became the world's first Open Source project, after Eric S. Raymond convinced the development team to use that term.

Now, plenty of pundits will tell you that the Open Source ecosystem has never been healthier, yet Richard Stallman's problems remain unaddressed. I am writing this article surrounded by computers - the one I'm working at, the one in the audio system it's connected to, the ones in my phone and my wristwatch; and, yes, just like Stallman, the one controlling my printer. Despite the apparent success of open source, *not one* of those systems is fully open for me to use, study, share and improve.

To me, Free Software has the hallmarks of a late-stage subculture: the scene has been popularised, the surface ideas taken and turned into mainstream businesses. All that is left is the hype: we are open! Buy our open thing! Another example: Scrum is currently such a popular anti-hero for showing how the Agile movement "went wrong" that Agile manifesto signatory Ron Jeffries complains about "Dark Scrum" and is even motivated to use scare quotes on the word "Agile". Undoubtedly, the people at that Snowbird meeting two decades ago were creators and fanatics, keen to "[uncover] better ways of developing software by doing it and helping others do it." But what of the legion of today's Certified Scrum Masters and Agile consultants? Do they all have the same deep understanding? Does such an understanding even *help* now, or is the context of today too far removed from the context of Snowbird in 2001 for the same principles and practices to be relevant? Are they geeks, MOPs, or sociopaths?

Regardless of the answers to these questions, the world's companies are not done being told that they are all software companies; that software is eating the world; that they are long overdue an "agile transformation" and that if they are failing it is because they are not agiling hard enough.

You pick up the allegorical, digital equivalent of a programming industry trade journal and read that your monolithic backend server

is yesterday's news. It's 20th-century technology. The new hotness is microservices - breaking each component in your server out into a separate deployable service with its own interface. This, they tell you, is Service-Oriented Architecture done *right*. Is that relevant? Are the benefits of separating the components worth the increased complexity of deployment, or the performance impact of so many API calls per external request? Only you and your team can answer those questions, but that does not mean that the authors of the article on microservices know nothing that is relevant to your situation.

Whether it's last decade's big idea or this morning's new Javascript framework, it was created by someone motivated to solve a problem they saw. Dismissing these things out of hand - the old because it is yesterday's news, the new because it is untried and immature - means missing out on fresh perspectives from which to view your work. As with many situations, application of empathy will go a long way. Empathising with the creators of the ideas or projects will help you to see what the world looks like from their position, why they solved things the way they did, and whether that is relevant to your own context. Maybe you agree that the problem they saw is real, but it isn't one that you have. If so, you have not found a tool that you can use - but you *have* learned more about your situation and somebody else's. And that understanding is where design happens.

Mainstream Is The New Hype

Adrian Kosmaczewski

Image: Verena Yunita Yapi on Unsplash

Early in September, while the first drafts of this article hit the administration console of WordPress, a friend of mine invited me to attend a projection of yet another Apple keynote, beer and pizza included; the usual hipstanerd package, in the office of some app development agency in the city of Bern. Surprisingly enough (or not,) Microsoft chose to host its online .NET Conference at exactly the same day and time.

A telling story of Apple vs. Microsoft, once again, clashing in my professional life, offering me another possibility to branch the future of my professional universe with a simple decision.

Let's start with the mandatory dictionary quote (after all, this is a license I can take, given that this is the first issue of this magazine.)

> hype | hʌɪp | informal
>
> noun [*mass noun*] extravagant or intensive publicity or promotion: his first album hit the stores amid a storm of hype.
>
> • [count noun] a deception carried out for the sake of publicity: is his comeback a hype?
>
> verb [*with object*] promote or publicize (a product or idea) intensively, often exaggerating its benefits: an industry quick to hype its products | they were hyping up a new anti-poverty idea.
>
> ORIGIN
>
> 1920s (originally US in the sense 'short-change, cheat', or 'person who cheats etc.'): of unknown origin.

In my own personal dictionary, I cannot avoid seeing Hype as anything else than a factor of stress. Not just another element of our developer culture, but an actual trigger of anxiety, burnout, Fear Of Missing Out (or "FOMO,") and other niceties that are plaguing the minds of my colleagues.

In many ways, Hype spreads like cancer. It starts often unnoticed, spreading and multiplying in otherwise sane hosts, and by the time it is discovered it is usually too late.

Hype increases job instability, it feeds anxieties, and sustains a rather shallow cohort of trainers, consultants, businesses, online courses and whatnot, venting platitudes and marketing dogma to whoever is worried enough to lose their jobs. It is a source of antagonism, decrepitude, and trolling, all of which our craft could genuinely avoid for the greater good. How many otherwise competent workers have been fired for not being "on the cutting edge" anymore? How many dollars were spent in training teams for technologies that simply disappeared a few years later? How many rewrites have happened to

take an otherwise fine working system into a new paradigm, dismantling teams, introducing bugs, and killing revenue?
And all of that, for what?
I have been victim of Hype many times during my career. Sometimes, I have to admit, I actually was one of the trolls feeding the anxiety of my colleagues with the technology choices I made. I even was one of the trainers, one of the consultants, and Hype has also paid my rent to a certain extent.
I can totally understand Hype as a phenomenon, and I can relate to its side effects. Then, who am I to judge? I fed it. It grew up on me. After all, the web has Hype in 1997. Later .NET was Hype in 2002. Also the iPhone was Hype in 2009. Docker was Hype in 2015. The four major technologies that have shaped my career, all touted as the world-changing things that they actually were. And at the time of this writing, Serverless is Hype and the cycle goes on.
As I said, I can understand Hype. But as a wise friend of mine once said, "the fact that one understands something does not imply that one has to agree with it." And Hype, I do not agree with it anymore.
For the sake of memory and history, here go more Hype-compatible terms from the past 30 years: CASE Tools, CORBA, RUP, SOAP, MDA, Software Factories, Semantic Web, OLPC, and (of course!) Augmented Reality, in both of its moments of glory, 2009 and 2017.
What is "Hype"? How can one define it?
Maybe I should define it through its opposite, through its counterpart: the "Mainstream".
If Hype is the future, the Mainstream is the past.
If Hype is unstability, the Mainstream is stability.
If Hype is fun, the Mainstream is boring.
If only it was so simple. For Hype is, maybe, just another acronym, like there are so many in our industry, one meaning *Hyperbolic Yet Passing Effusion*. Of course, we all know *People Can't Memorize Computer Industry Acronyms* or PCMCIA - yet another Hype technology back in the 90s.
(The astute reader will have noticed the absence of the word "Legacy" in this text. It is no coincidence, as once again, that word is more closely related with subjective visions of evolution than actual facts. Otherwise, go tell that poor soul working in a datacenter near Zurich Paradeplatz that their COBOL banking system, running flawlessly and profitably since the 70's, is "Legacy.")
Should we avoid Hype? This author says yes, for sanity, although it is very simple to realize that for good or worse this cannot be done; we simply cannot fight against it. We will not be able to make it disappear. Can we fight against Vogue, Versace, Karl Lagerfeld and Anna Wintour all saying unanimously that, let's say, blue garments are the new "it" this season?
Hype is very often related to fashion, a concept itself related to taste, a concept subjective by definition. It is actually funny to consider that Hype is highly subjective and irrational, given the tendency in geek circles to value objectivity as the guiding rule for all decisions in the world. Yet, more than once, a technical decision is simply guided by this angel of death called Hype, flying its wings once again in a meeting room somewhere.
(The irony of the subjective/objective dichotomy does not stop there, as the heated discussions about "Programming as Art" tend to show. But this will be the subject of a future edition of this magazine.)
Maybe the problem is that Hype is devoid of

substance; is it the case, though? Can one always discuss Hype in rational terms? If Hype is fashion, and thus subjective, and thus irrational, can it contain substance?
Clearly, history shows that yes, Hype can sometimes be full of substance, with more or less the same probability of being full of excrement (the editors of this magazine are keen in using the most appropriate language to avoid hurting any susceptibilities.)
What is then the magic substance that makes Hype stand the test of time? Is it *Market Share*? Is it the availability of *Conferences* and *Documentation*? Is it a healthy mix of FOSS (Free and Open Source) and Commercial projects around it? Is it a large number of answered questions in Stack Overflow with a score higher than 1400?
Can Hype become Mainstream? Of course it does. C++ was Hype in 1990. JavaScript, too, will become Mainstream one day - and the author of these lines thinks that this will not take long now.
Instead of fighting against windmills like the Quijote, here is a wish: can we stop feeding Hype and prevent it from spreading like a cancer in our production systems? Could we stop using Hype-compatible technology for our next application server, mobile app, healthcare product, and instead insist that proven, old, Mainstream, boring technologies be used instead? The author of this text hopes so, and fervently for that matter, that one day sanity will guide the CTOs of the world into the creation of the next *disruption* using a 30 year old language, and a framework recently updated to version 19.

(To be honest, this very blog encompasses that vision, using a decidedly Mainstream - slash boring slash stable - platform like WordPress, supported by the venerable quartet of PHP, MariaDB, Apache, and FreeBSD. But I hope that the mention of PHP in these lines will not alienate our readership.)

Hype *can, should and must* be kept away from the production process of critical systems. This will not always be the case. But one can always dream of a better world.

In the meantime, I chose to attend the .NET Conference - and I do not regret the choice. Being 18 years old, C# is mature enough now for being considered Mainstream. Even the "classic" .NET Framework has been declared Mainstream - or was it Legacy instead? Regarding my friends, well, I will have a beer with them tomorrow anyway, to celebrate the release of this very magazine.

Issue II

Quality

November 2018

The Ipsum Quality Process: 12 Steps To Better Teams

Adrian Kosmaczewski

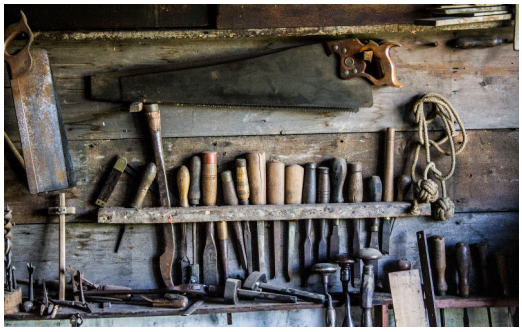

Image: Philip Swinburn on Unsplash

Back in 2005 I had an interview for a job in a software consulting company in the French-speaking part of Switzerland. The interview in itself was dreadful; the hiring manager was certainly more interested in the "gaps" in my résumé than anything else. But one thing drew my attention, and made the interview nevertheless memorable; this person boasted that their company was one of the few "certified CMMI level 2" in Switzerland, and that they were in the process of preparing for the "level 3" certification.

To be frank I had no idea what this person was talking about; so I looked online and discovered that:

> Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program. (...) It is required by many United States Department of Defense (DoD) and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization.
>
> - Wikipedia

Methodologies

The important word in the quote above is "claims." CMMI is just another quality framework, just like ISO 9000, Six Sigma, Total Quality Management, or even the Joel Test; the primary job of these frameworks is to provide a nice (and sometimes very expensive) badge for companies to feature in their website, getting C-level managers to tap each other on their shoulders and give themselves another huge bonus.

A quality framework does not, per se, provide any insurance against big balls of mud. It does not establish any expected level of quality. It provides no insurance against impossible deadlines, crazy customers, developers with family issues, computer failure, misleading vendor marketing, or anything else. Yet, companies all over the world subscribe to these and other frameworks, creating an industry of speakers, consultants and conferences worth billions, yet unable to answer The Real Question That Everybody Asks About Their Software Project™®©:

> Will the project be finished in time, delivering the features the user needs, and within budget?

If these frameworks are of no use, what is then, the magical factor that separates the (few) companies releasing quality software in time (there are, believe me) compared to

the rest? Why do most projects end up with a Death March instead?

About Quality

I tend to describe Quality with a simple phrase, almost a mantra; a phrase very easy to remember, borrowed from a book somewhere, which I will shamelessly transcribe here for you to invoke *ad nauseam* in your next meetings; répétez avec moi:

> Quality is doing the right thing,
> and doing the thing right.

(As the astute reader has guessed by now, the quality frameworks discussed in the previous section only care about the second part of the statement, if anything.)
But what is the *thing* we have to do, and rightfully so? Well, in the case of software engineering, that "thing" is *not* software - and I am pretty sure I have just made a few eyebrows raise in perplexity. No, ladies and gentlemen: the "thing" in question is the following:

> To take care of your team.

There you go. Here is my invoice. Welcome to the light. There is no other thing to know. *Quality is taking care of your team.* A product with good quality is simply a side effect of a team being taken care of (sorry for the functional programmers among my readers.) Do you want quality in your software product? Take care of your team. (Hey, Graham, we should start a software consultancy and copyright this idea fast.)

The Ipsum Quality Process

And what is, exactly, taking care of a team? I am glad you asked. Taking care of a team can be defined as a set of 12 simple bullet points that you can start following right now in order to make better software:

Hire inclusively

Having members with disabilities is important for your team - if anything because, among other reasons, it can help make your product more accessible.
Having team members from other cultures is important for your team - if anything because, among other reasons, it can help make your software multicultural.
Having people with various ages is important for your team - if anything because, among other reasons, it opens up opportunities for coaching and mentoring.
Having people from all genders is important for your team - if anything because, among other reasons, it will make the world a wonderfully better place.
All of this is important for your team, and hence it is important for your product, for the industry, and for the world.

Pay your engineers decent and equal wages, including extra hours

The software engineering field is extremely profitable, and it is not unheard of for a consulting company to have margins well above 30 or 40 percent. Why is it then that only managers get bonuses at the end of the year? Why do not these companies pay extra time? Why is it that the salaries of software developers are going down?
One of the main reasons of this situation is the lack of unions in our field; the bargaining power of software engineers is scattered, atomized, which was exactly the objective of

the managers of businesses in the first place. As a developer, you must ask for a clear, open salary policy, and make sure all engineers are being paid in equal terms in correlation to their experience, regardless of age, gender, nationality, skin color, or any other attribute.
Regarding extra time, if it must happen for any reason, *pay it* and then *compensate it* with extra days for rest (see "plan for 6 hours of actual work per day" below.)

Stop cramming people in open spaces

Get private offices for your team, with at most two or three people in them. Forbid the use of smartphones and Skype and Hangouts in those rooms; they are there to be used for those three or four hours of deep concentration that make projects move forward fast (see point 9 below.) Set up many team rooms with whiteboards so people can discuss and talk. Isolate all of those rooms acoustically from one another.
You need to coordinate, yes; but not as often as you need to get things done. Do not say that private offices are more expensive, because they are not - at least not in the long run. Have a broader mindset and stop telling lies.

Do not offshore projects

Offshoring software projects, that is, outsourcing them overseas, is the 21st century equivalent of economic imperialism, keeping countries in Latin America, Asia or Africa earning peanuts while agencies fill their pockets, and blocking the development of those countries. Emerging economies should be building their own products, and developed economies should pay for those products. This is the way to development; not imperialism.

All countries need software engineers to stay where they are, if anything because their salaries feed a greater economy. And also, the trouble and the mess of managing a project overseas is actually going to eat all the benefits of having a team located next to you, so do not even think about offshoring.

Train teams in old and new technologies

Give everybody in the team a certain budget to spend in conferences, online courses, and reading material. Make a library of physical books and stack it with some timeless classics. Give your team clear learning objectives, such as getting certified, preparing a 20-page document to share with the team, or talking at some event, and set up bonus structures tied to those requirements.
Make them teach each other all of those new things, continuously; either in pair programming sessions, either during code reviews, or in small 20 minute talks in the conference room in front of the whole team (which is a good exercise for people new to public speaking.) Review that your team has learnt and look up to the horizon. Rinse and repeat.

Drop the foosball, ping pong, pool table, karaoke, game console, ...

Seriously. It only shows that all you know about developers is what you learnt watching episodes of "The IT Crowd." Those things are not only annoying, they are vexing and demeaning.
Build a library instead. With books, you know, the dead tree kind, and some coaches; a water boiler and some tea; quiet music, because relaxation is key. Offer a place for relaxation and your team will love you. Peace and

thoughtfulness will make your team create better software. Certainly not foosball tables.

Watch for mental health issues in your team

Mobbing, burnout, harassment, depression, fatigue; all of these things are happening right now in your team. Oh yes, they are. Are you even paying attention?

Watch for body health issues in your team

Does your team have standing desks? Do they have good lightning? Do they have air conditioning in summer and heating in winter? Do they have to use noise-cancelling headphones every day? Do they have good monitors to prevent strain in their eyes? Are they sitting in comfortable chairs? Do they take enough pauses? Do they use ergonomic keyboards? Are they doing extra hours? Are they eating in front of their computers? What are they exactly eating?

Plan for 6 hours of actual work per day

Each workday should be divided as such:

- 4 hours a day of actual work (coding, writing documentation.)
- 2 hours a day of mingling, drinking coffee, lunch, meetings and coordination.
- And 2 hours a day for learning something new.

More than 6 hours of actual software development work per day is a recipe for disaster, burnout, and turnover (been there, done that.) If your work laws mandate 40 hours a week of work (which is the case in Switzerland,) make sure that those extra 2 hours per day are spent learning something new instead of writing more code (see "train teams" above.) Send your team members home at the end of the working day. Turn off the electricity in the office if needed, so that they actually leave. Compensate for extra time (see "pay decent and equal wages" above.)

Embrace remote work

Remote Work Works™®©. And beautifully so. Our field is perfectly suited for working from home. Embrace it and drop the requirement for command and control. You are going to make economies, and your employees are going to love you more for this.

Do not be a project management methodology fundamentalist

You do not need to have a scrum standup meeting every morning. You do not need to have that retrospective every two weeks. You do not need to follow the PMI workbook to the letter and punish those who challenge it. Doing so is nothing else but cargo cult. Remember this:

> Waterfall has put a man on the moon. What has your methodology done so far for your team?

Change your approach to job interviews

Stop asking about round pot hole covers. Stop those live coding challenges to reverse a linked list. Drop the academic requirements. Instead, hire for character first, and then for skills. Of course you need to be sure if the candidate can do the job, so here is a simple technique: *hire them to work in your team for a day.* Just a day. Pay them a full, fair freelance rate (see "pay decent and equal wages" above) and watch

them evolve in your team. At the end of the day, ask yourself and your team these questions:

- Were they good at reading code?
- Were they nice people?
- Did they suggest improvements to the code?
- Did they ask good questions?
- Did they behave ethically?
- Did they say "I do not know" often enough?
- Were they humble?
- Did they show empathy and kindness towards other team members?
- Did they looked happy being there?
- Did they ask about code coverage or the CI setup?
- Did they offer to help, even if the time was short?
- Did your team come back to you with big smiles asking you when they were going to see them again?

If most answers were "yes," just hire the person.

Disclaimer

If you are a project manager or software CEO reading this article, you must be asking yourself the following question: does the Ipsum Quality Process guarantee the outcome of my project?
No, of course not. You can still have incompetent people making bad decisions, even if they are healthy, well paid, and enjoying excellent work conditions. But at least you will have reduced your employee turnover; you will spend less money in hiring costs; employees will have less sick days; they will be proud of working in those teams, and will attract their friends; they will end up doing everything they can for the company, including, why yes, writing high quality code yielding high quality products. And I am not the only one who thinks like this.

> Take care of your employees and they'll take care of your business.
>
> - Richard Branson, Founder of Virgin Group.

In short: only after your company fully implements the 12 steps above, we can sit down and talk about TDD, Agile, CI and whatnot.

Conclusion

To finish the anecdote that started this article, I heard the company (which shall remain nameless) went almost bankrupt, after taking in a project much bigger than they could chew, having it developed by an offshore team working in an open space. Most of their technical crew left the company less than half a year later. I read that they struggled to survive, and they ended up sacking most of their management and enduring a deep restructuration.
As for the CMMI level 3 certification, a quick look at their current website tells me they never got it, or they just do not care about it anymore.

> "Move fast, break things and piss off the ones taking over" ™©®
>
> - Eliezer Talon

The Various Meanings of Quality

Graham Lee

Image: Dane Deaner on Unsplash

What, to you, is quality software?
I'm sure that I will get different responses from different people. Quality is not absolute, and what you consider high-quality may be irrelevant to somebody else. Your view of high quality may even be a sign of poor quality to another reviewer. Let's take a look at some aspects of software quality.

Code Quality

Developers, myself included, often think of the quality of the software's source code: is it easy to understand and change? Does the code use paradigms that I find admirable? Does it use functional programming, for example, if I'm in to that sort of thing?
Did the author write automated tests? Do the tests provide good coverage? Are they appropriately distributed between different levels of granularity? Do the "grains" (the units, modules, or subsystems) represent useful models or abstractions?
Is there documentation? If there is not, is it because the code is self-documenting? Is there a single command to type or button to press to get a production build out? Does committing to the repository automatically trigger a production build?

Does Code Quality Matter?

These questions all relate to a particular perspective on the quality of software, but we must be honest with ourselves. For the most part, customers considering whether to license a particular software product or subscribe to software as a service very rarely even *look* at the source code, let alone make an informed judgement as to its quality. In most cases, they aren't even allowed to look.
Even in the case of free software or open source products, the likelihood of spy-before-you-try is low. I cannot bring to mind the thousands of such projects I must have used during my career to date: Eclipse, LibreOffice, Linux, the GNU Compiler Collection, Darwin, clang, LaTeX, Firefox, FreeBSD, GNU Emacs, node.js...and I can't think how vanishingly small must be the fraction of those of which I have even *partially* examined the source code. Probably not even one per cent.

Lemons

Now that means that if the quality of a software product, or service, *is* related to the code quality, then there's a lemon market for software. People can't (or don't) evaluate the quality of the code, so won't pay more for premium-quality code (and hence software) than they would for a clunker.
(As an aside, the solution for the lemon problem in motor vehicles is to provide a warranty. In software, commercial and free software alike is provided "AS IS", all in capital letters,

with no suggestion that the maker expects it to work the way you like, or even the way they said in the marketing materials.)

Zen and the Art of Software Maintenance

The quality of *the code* in a software system is an aspect of its internal quality. Borrowing an analogy from Robert M. Pirsig, interest in this form of quality is like being interested in the internals and maintainability of a motorcycle engine. It's a "classical" form of quality, in which one must be mindful of all the workings and details of the system under consideration. Other people will have other ideas of software quality, perhaps informed by the "romantic" school of quality.

To Accessibility and Beyond

Is the product accessible? That may mean that the controls are suited to full keyboard navigation, text-to-speech readers, joystick control, or other adaptive input and output schemes. Accessibility also includes ensuring that the colour scheme used through the UI is suitable for people with colour vision deficiency.

Some would say that accessibility is one aspect of the wider principle of *usability* and consideration of the user experience. Can the capabilities of the software be discovered easily? Can they be used efficiently? Do the capabilities that a given person requires match their expectations? Do those capabilities even *exist*?

Other External Qualities

We could carry on listing other external "qualities" of a software product or service: security is a very big topic and indeed the next issue of de Programmatica Ipsum will focus exclusively on that.

Performance, both in terms of speed of computation and responsiveness (a researcher running their scientific codes on a high-performance computing installation will experience very high latency, as the queueing system could take days to even start their job; they may not think of this as "unresponsive" if submitting to the queue happens quickly). The resource usage is another form of quality, as is the clarity of the customer documentation or even the helpfulness of the customer support team.

How We Plan For These Qualities

Many of these quality attributes will be described by most people on a software project as the "non-functional requirements" and batched together as things we should probably keep an eye on while planning to construct the functional requirements. Meanwhile, the software architect (if there is one) will be trying to design a system that satisfies all of the non-functional requirements while paying half a mind to the question of whether it can satisfy the functional requirements.

Which finally, nearly eight hundred words into an article on software quality, brings us to the elephant in the room: does the software even *do* the things people want of it? That seems like an important measurement of software quality, too.

The Quality of Correctness

Does our software do what our customers want, or need? Manny Lehman tells us that in many cases the answer to this question is not stable. You are not making it up and your cus-

tomers are not deliberately being awkward, their needs genuinely are evolving. Perhaps, in a very postmodern way, the trigger for their evolution was the introduction of the software itself. The availability of version 1 freed up some time, so that they realised they could do these other things if only they had version 2.
Given the evolutionary nature of software requirements, how should we consider the "quality assurance" function in a software team? For a start, let us rid ourselves of a misconception: your "QAs" do not assure quality. At best, they *measure* quality by testing (or creating automated test scripts for) your software. But testing does not assure us of quality. As Dijkstra said, it can tell us when bugs are present, but not when they are absent.

Regression Tests

Most of the software projects I've worked on have had a battery of "regression tests". Only some of these were written to detect regressions. Most were written while TDDing, or to demonstrate that a feature worked as part of that feature's construction: they are accretion tests, rather than regression tests.
A *true* regression test would be one where we know that the behaviour must not be reverted. But that doesn't mean that we should test *anything that was ever added*, it should mean knowing *what your customers are using today* and how bad it would be if it were to break.

"Enough" Quality

Notice that I said *how bad*, as there are different amounts of badness. The Joel Test asked whether we always fix new bugs before we write new code. That doesn't seem like a good way of working. Success in business is always about opportunity, and so we need to consider *opportunity cost*. If three people would be put off our product by a minor bug, and thirty people would buy the product if we added a minor feature, we *should* write the new code before fixing the bug. On the other hand if the bug breaks every day features for every customer of our product, then yes, it's time to fix it.
There are lots of ways to evaluate the quality of a software product. Now what, to *your customers*, is quality software?

On Some Things Quality Related

Ioana Porcarasu

Image: Brian Goff on Unsplash

Through the eyes of the writer

You never heard of me before, as this is my first article that "saw the public light". I don't have years and years of experience on my shoulders, but I can give you my thoughts, my personal opinions, and it's totally understandable if you don't agree with something, or even anything.
I started my journey into software engineering as an Automation in Test Engineer, around 7 years ago, and at that stage I had no idea what a complex concept quality was. During Computer Science university courses, we kept learning basics of programming languages, OOP, logic, maths, but barely close to nothing about what it involved into a product lifecycle. How do you get from one idea into an entire product available to customers, with a whole system on its back to monitor and track the users behaviour, to learn from their journeys and add improvements to it? And where is quality standing in these workflows? When are you confident enough in its capabilities to say: "Yes, we can ship this. Customer can have it!"?
I'm writing this article and keep re-reading, re-editing and re-doing it. Every time, I keep adding something, a small improvement hopefully, but no one else had read it except myself. All I do is updating it all over again, but with no external feedback. So how can I know that my "product" meets its minimum level of quality? How can I know it meets your needs and expectations?

So, what is quality?

The most basic definition of quality can be identified as grading how good or bad something is. But that doesn't necessarily mean that "your bad" is also "my bad" , or the other way around! Quality is a subjective attribute, with different meaning to different people.
Quality, in essence, is embedded in our everyday routine: we are looking to buy a new gadget, so we search reviews for different products to asses their quality before actually purchasing one. We read positives and improvement points for different brands until we have the confidence that one of them can actually meet our needs, our expectations. We are researching for our next holiday, so we browse countless websites and brochures until we find the one that fulfils our requirements. So our entire live is actually rolling around this concept of quality.
And this can be seen in product development as well. We make assumptions all the time related to what we think is right for the customers. But quality should mostly be seen through the eyes of the customer.
Software quality is defined as the level to which a system meets specified requirements or user needs and expectations. Testing has become an important segment in the software development process to ensure its quality.

One of the basic testing principle is related to verification vs validation. Verification is testing that your product meets the requirements written for the product: "Did we build what we said we would? Did we build the system right?". Validation, on the other hand, tests how well you addressed the business needs that caused you to write those specifications: "Did we build what the customer actually needs? Did we build the right system?".

Ever since the oldie, waterfall methodology, we understood that we didn't actually provide quality services. We could work for a year on a new product; research done, implemented this awesome new idea; tested and validated it for a couple more weeks, and finally delivered it to our customers in a shiny new box with a red bow at the top; if the product is not fit for purpose for the customer needs, what can we do next? Can we say anything about the quality of that product? On one side, we verified it extensively during our (let's say) 6-week window and reported appropriately, it met the requirements and we were highly confident in it! We build the system right, but coming back to the previous point, was it actually the right system? Even if the idea might've been validated initially, in the year of developing and making the product available to the customer, he might've changed their priorities, he realised that actually another thing is needed, another problem needs solving . So, providing quality enforces this idea of building the right system for the user. Feeding back to the customer more often was essential to understand if we're providing quality services and eliminate waste.

And that's one of the reasons why software development evolved into all the new different trends happening currently.

One definition proposed by Jez Humble is that DevOps is "a cross-disciplinary community of practice dedicated to the study of building, evolving and operating rapidly-changing resilient systems at scale.". Continouous delivery concept is also seen as "the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way." The entire community understood that we can use different techniques, we can collaborate more and we can help each other to mitigate risks and get the confidence that we are delivering value to our customers, and all this through small incremental changes.

We are more focused now in working in cross-functional teams that broke down the silos and the fences, and we can give feedback as early as possible to improve the quality of the system. Just by asking questions when analysing a set of requirements challenges the group of thinking at different scenarios, improving collaboration between developers, testers, and operations. And how much does that cost? Nothing! We didn't even start implementing it!

And quality doesn't cover only the system anymore. It evolved as a whole new process, and we moved away from testing only roles, but to Quality Engineering ones. And as the Testing manifesto is stating, we are not focusing only in proving a level of confidence for the product anymore, but we actively collaborate towards:

- *Testing throughout OVER testing at the end.* As we've seen in so many examples and memories, the amount of waste increases exponentially if we leave the testing at the end of the product lifecycle. Instead, we should focus our efforts in

testing early and testing often. And with the massive help of automated checks, it all comes back to fast feedback loop!

- *Testing understanding OVER checking functionality.* Similar as above, why waiting to get a new build to test and confirm behaviours, when some of the possible issues could be found in earlier stages? Shifting left the testing is a massive improvement to the overall quality of the system as it might raise questions in team's understandings or even customer expectations. Building a common understating will automatically lead to improved quality.
- *Preventing bugs OVER finding bugs.* This is essential if we want to respond fast to our customer needs. We need to think more about the "life" after releasing to the customer. How can we prove it is actually working as expected and if anything happens, we can act fast?
- *Building the system OVER breaking the system.* We need to be proactive and work closely with the engineers, POs to identify what the customer actually wants. We need to steps into theirs shoes and help understanding the problems we are trying to solve for them and find the best solutions.
- *Team responsibility for quality OVER tester responsibility.* Every single team member is responsible for quality of the product we are building.

Testers do not provide assurance anymore; they help with analysis and feedback. With every engineer writing and running tests with their commits, they show ownership. With the whole team caring about the state of the system or of the build, they show ownership. With good monitoring and logging in place, the team then owns quality. Through a continuous-improvement mindset applied throughout the lifecycle of a system, we demonstrate that ownership of quality. And the truth is that shared ownership results in higher quality!

Issue III

Security

December 2018

On Modern Security Culture

Graham Lee

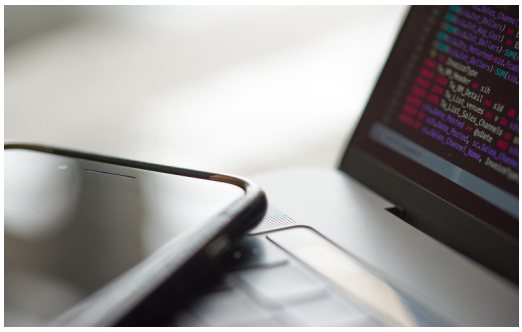

Image: Chris Ried on Unsplash

Long before Adrian and I started *De Programmatica Ipsum*, we met at a conference in London where he had invited me to talk on mobile application security. Reflecting on this issue's topic, I compared that experience with the infosec-focussed events I have attended. I thought of the worst aspects of the infosec culture that I've seen. Writing about those would make for a full and interesting article, but wouldn't be particularly inspiring. Instead, I'm going to talk about some of the more progressive parts of the infosec world, by introducing the people who informed my view on those parts. If they inspire you as much as they did me, then you will come to think of infosec as something your whole organisation, and your whole community, can come together on.

Window Snyder

Window Snyder is currently Intel's Chief Software Security Officer. She has a history of transforming software security at computing's big names: Apple, Mozilla, and Microsoft are all former employers. At Microsoft, she was security lead for Windows XP Service Pack 2 and Windows 2003.

Trustworthy Computing

These products were released in the wake of Bill Gates' Trustworthy Computing memo.

> Today, in the developed world, we do not worry about electricity and water services being available. With telephony, we rely both on its availability and its security for conducting highly confidential business transactions without worrying that information about who we call or what we say will be compromised. Computing falls well short of this, ranging from the individual user who isn't willing to add a new application because it might destabilize their system, to a corporation that moves slowly to embrace e-business because today's platforms don't make the grade.

After that memo, the company stopped development of its products to train development teams on software security and made security and trustworthiness top priorities.

Threat Modelling

Microsoft's approach to security during that time was shared widely. Their approach became the basis for Secure Development Lifecycles (SDLC) across the industry. Snyder's co-authorship with Frank Swiderski of Threat Modelling is an important part of this outreach. I believe this book shows how a modern, self-organising software product team can make itself security-infused. There's

still a place for external consultants, penetration testers, and dedicated security teams. These are people who _inform_ and _educate_ the product team. The product team are ultimately aware of their product's security model, its risks and how to address those risks. If your product team is not sure how to think about their product's security, get this book and run a workshop. If you and your team do not think security is your problem, then you _definitely_ need to read and internalise the Threat Modelling book.

Kate Moussouris

k8emo is also a Microsoft alumna. It's time to revise any opinion you have that Microsoft is not an innovative software engineering house. She has advanced the way that organisations including Microsoft and the U.S. Department of Defense interact with external security researchers and testers. Moussouris was instrumental in introducing a "bug bounty" program at both organisations, and elsewhere. A bug bounty is a prize paid to an external reporter of a security vulnerability. Bug bounties are one incentive designed to support responsible disclosure, another of Moussouris's successes in the field.

Responsible Disclosure

The actors involved in security vulnerability discovery have different, conflicting objectives. The product vendor may prefer never to admit that their software was insecure. They would like to silently patch their product (if they intend to patch it), and move on as if nothing had happened. If an external security expert found the bug, they will want to publish details and get credit for the discovery. They may use that discovery in their marketing materials, as demonstration of their expertise. Unfortunately another group who would benefit from public disclosure of security bugs is the attackers. They can use information about the bug to design an attack targeting people who haven't yet remediated the problem. Which leads us to the customers, the people using the software. They want to know about the problems so that they can patch, or change their security policy, or otherwise address the risks associated with the vulnerability. Responsible disclosure builds a compromise position from this conflict which represents the best trade-offs for all parties except the attackers. The person who discovered the flaw reports it privately to the vendor, and negotiates a timeline to public disclosure including full credit. The vendor gets time to fix the bug and get the fix to customers before public disclosure. Users get told about vulnerabilities, but only after the vendor has issued a patch.

Bug Bounties

Bug bounty programs act as a form of virtue signal for responsible disclosure. It's important to see that bounties are not a _replacement_ for Threat Modelling and internal security practices. Instead, bug bounties indicate that a vendor acknowledges that external experts may find problems they have missed. A company with a bug bounty program is saying that it supports responsible disclosure, and will reward others who cooperate in its responsible disclosure approach. The bounty is not paid to _anybody_ who finds a security bug. Rather, the vendor pays those who engage in its responsible disclosure process. The vendor hopes that the value of the bounty payout

stops the researcher from publicising a vulnerability as soon as they discover it. That situation is called a "zero day" or 0day, because attackers have the opportunity to exploit the problem on day zero of the disclosure and patch timeline, before the vendor has a chance to fix it.

Modern Security Culture

Through practices like threat modelling and responsible disclosure, information security has moved from a combative to a collaborative art. We no longer have a security team or an external penetration tester telling us "no" after we've built a product. We have devops and devsecops: a culture in which a software team works together and treats security like any other attribute of their product. Such collaboration extends beyond the org chart, with multiple parties coordinating their vulnerability disclosures to the benefit of the people using our software. The work of Window Snyder, Kate Moussouris and others enables this cultural shift, helping our whole industry to make computers safer and more helpful.

The Weakest Link

Adrian Kosmaczewski

Image: Tan Kaninthanond on Unsplash

At the end of 2000 I was studying IT management in the Universidad de Buenos Aires, and the teacher in front of my class was trying to instill some basic notions of computer security in the heads of a rather skeptical (albeit ignorant) crowd. Out of a certain exasperation, said professor stopped the class. He asked one of the few lucky owners of a cellphone in the classroom for his phone number. After entering the number into a device looking like a handheld calculator, he asked this unsuspectful student to go outside the classroom and call a friend or a relative. The student complied. After a few seconds, to our astonishment, the machine picked up the call. We could all clearly hear our friend talking to his mum about dinner plans for that evening. I was shocked. I had no idea how cellphone networks worked. But I was under the absolute assumption that privacy and security were given attributes of them. That whoever designed and built that infrastructure, had them as major goals. Was not private correspondence protected by the constitutions of most countries in the Hemisphere, after all? Why would cellphone communications be any different? Was this assumption foolish?

Rude Awakening

A few months later, I bought a copy of "Hacking Exposed 2" by Scambray, McClure and Kurtz; it was the only book about computer and Internet security I could find in the shelves of the stunning and recently opened "Ateneo Grand Splendid" bookstore in Buenos Aires. (It must also be said that this is one of the books that changed the meaning of the word "Hacking" forever in the minds of millions, a confusion that Richard Stallman himself has tried to debunk, but to no avail.) In that book I learnt that most of the traffic in our networks back then was completely unencrypted. Which meant that, with the proper setup, it was easily "sniffable."

First Steps

Out of curiosity, I used this brand new search engine you might have heard about, "Google," to learn more about the subject. I found a copy of CaptureNet, a freeware packet sniffer part of the SpyNet/PeepNet by Laurentiu Nicula; then I looked up for the port number used by MSN Messenger (it was 1863 in case you were wondering.) Finally I found out how to enable "promiscuous mode" in the network card in my laptop. I plugged all of these pieces together in silence, my coworkers fully unaware of my proceedings. In the small LAN of the office my company had north of Buenos Aires, we were all using Windows 2000 Professional there. And, as it were, it turns out we were using a good old router, not a switch, which undoubtedly helped me fulfill this task. I finally turned the sniffer on. Instantaneously, my screen started to show me the conversations my peers were having on MSN Messenger. And I mean all

of it. Every detail of their private lives, the current business deals, the comments of the latest news. Every "smiley" they were sharing. Everything in their private lives, every single word they said. All on my screen, ready to read, without any encryption. After changing the sniffing port to 80, I used CaptureNet's uncanny feature of reconstructing the web pages. All of my colleagues browsing at that very moment, including images and scripts, appeared in my laptop. Those sessions shall remain anonymous and forgotten. I got so scared that I basically hit the stop button on the sniffer and deleted the logs. For the first time in my career (I had been working as a software developer for 3 years so far) I had the sensation that everything we did in our industry was extremely fragile, insecure. It seemed to me that we were all blissfully unaware of how naked we were. It was even worse than watching our teacher listening to private phone conversations. This was even simpler and cheaper - no need for custom hardware.

Wannabe Cracker

This new knowledge took me to a rather somber hobby, one of which I am not really proud nowadays. Around 2002 I got into the habit of scanning random IP ranges on the Internet, finding computers running Windows 95 or 98 with port 139 (NetBIOS) wide open, and then connecting to them using Back Orifice. Connected to those machines on the other side of the planet, I watched live. Those users were typing documents, filling spreadsheets, or browsing the web. I would then take over their mouse, open a Notepad file, paste a pre-written text explaining to them that I meant no harm. Explaining them that their system was open to intrusion, and how they should protect themselves. I would then leave without further action, a shocking small Notepad window behind me. A mirror of their own nakedness, and most probably a gaze of terror in their eyes. In my defense I will say that I never, ever made any change or stole any information from those machines; my intent was to raise awareness, although I reckon my methodology was probably not the most adequate or tasteful. Security breaches like these were by then already the matter of urban legend. Or worse, heated discussion around an almost expected feature in every new version of Windows. The situation of Microsoft regarding security was so serious that Bill Gates himself wrote the now legendary "Trustworthy Computing" memo to his employees, making the book "Writing Secure Code" by Howard and LeBlanc a mandatory reading inside his company. The motto of Microsoft was "a computer in every desk." Unfortunately said motto made no mention of how secure that computer had to be. This is to me another proof that "moving fast and breaking things" is one of the most harmful policies a company can follow.

The Dawn Of A New Era

There was a bigger question that popped into my mind back then. I was just a software developer without any kind of formal training, with just a basic amount of curiosity and a little experience. Yet I could put together all of those pieces in an admittedly standard computer. What could then be happening in governments or companies? What would be the level of intrusion of these companies in our lives? How much did our government and corporations know about us? Well, in the case of the Argentinian government, not a lot. Its

level of competence in IT was still a long way off (one could argue that ignorance was a bliss for the Argentinian people back then,) and there were other problems to worry about. But of course I figured out that great powers like the USA, Russia or Europe were easily reading (or at least storing) everything we said and did online. Because doing so was not only cheap for them (I had not spent a single dime in my setup, using a company laptop and some freely available software) but also extremely convenient and strategically important. Actually, it made more sense for governments to actually record everything that was going online and storing it in a database, than not doing it at all. All things considered, having all of that information in a database for later evaluation was better than not having it. Turns out I was right, and yet, still very naive. I had not realized that nuclear plants, hospitals, pacemakers, finance and nearly everything that uses electricity, was already being managed with computers connected to the Internet. "Geekonomics" by David Rice would not be published before 2008. Bruce Schneier blog was still just a newsletter. And IoT was... well.

The World We Built

We, designers and users of the technology of the future, eager to use the latest gizmos and the newest of approaches, we were feeding a silent surveillance machine, the product of which, 18 years later, is the slow establishment of the largest coalition of fascist leaders in modern history. As citizens, as technology designers and users, we all have contributed to the growth of this machine and the birth of this new world order, through none other than Facebook, MSN Messenger, Skype, Twitter, Tumblr, iOS, Android, Google, and so many other companies and systems. But we can change this, the same way we unconciously decided to make this happen. And of all citizens, software engineers and IT experts have an ethical duty to make computers secure, acting in two fronts at once:

- First, by designing, implementing and deploying systems in a secure and privacy-conscious fashion.
- Second, by teaching and raising awareness of the various issues around security and privacy in our modern infrastructure.

Humans, and particularly those working with computers, are the weakest link in the chain of security.

The World We Could Have Built

We are the ones giving out our (weak) passwords when receiving a phone call from the "support" team of a company whose products we use, or when a stranger asks us for it on the street. Clicking "I Agree" on most privacy agreements without even skimming the text. Not (fully) understanding how encryption works, and why some algorithms are of no use today, and even better, rolling up our own without first reading Schneier's Memo to the Amateur Cipher Designer written in... 1998. Not even trying to configure PGP in our mail clients. Designing purposedly complicated systems that others cannot properly configure, leaving security holes open. Forgetting to add SSL certificates to that new website we built for a relative. Misconfiguring routers and firewalls for convenience... or plain ignorance. Storing passwords as plain text in databases, and then, God forbid, sending them to users via e-mail. Not even trying to explain

our pointy-haired bosses that this should not happen. Ignoring two-factor authentication in our accounts. Letting ourselves be easily spoofed by politicians, as they try to convince us how weakening encryption is important for our "security."

Our Duty

It is our duty, our ethical duty, the one with utmost importance, to realize that we are humans, and that we, the people, are the weakest link in the chain of security. The good news is, this link, however weak, has an uncanny capacity to learn and change. We can change this world we built. We can stop building the current one, right away. Let us build a world where we protect each other, consciously, all the time, and where the systems we build serve this purpose. A world where privacy and security are basic human rights. Let us teach each other how to build this new world, like my visionary teacher tried to.

Secure Development Is Dead, Long Live Secure Development

Anastasiia Vixentael

Image: Nick Boyer on Unsplash

Does not it feel like the world is on fire? Security talks and blog posts usually start with horror stories - for instance "application security is important because without it you will be hacked here and now." But I have had enough of this doom and gloom on my Twitter feed and mailbox, and would like to talk about something else. Only those who (happily) live under a rock may have missed the latest news. One company has a critical security bug, another has leaked emails and passwords to millions of user accounts *(yours and mine included,)* and another will be fined millions of Euros due to a GDPR violation. I do not want to add to the existential dread, but I would love to discuss the problems behind creating secure software.

Why Do We Care?

Thirty years ago, before the Internet became widespread, only military and governmental services were interested in data protection. Back then it was as complicated to gather and analyze data as it was to steal it. Today more than 1300 new apps emerge in app stores every day; most of them collecting, processing, and transfering data - risking its confidentiality and integrity at each step. Let's put on our app developer's hat. Risking our customers' data, those who trust us to handle it with great care, is more than a question of it being "good or bad." Why? Because of the lawsuits, fines, and other more terrifying consequences that may loom over the business that we build, or are employed by. Competitors or attackers will be thrilled to discover you've left them an entryway, too. Protecting users' data and privacy is not only a sign of product quality. It is also a way of showing respect for those users and their rights. One could argue that data security has a "a negative value", meaning that it requires a lot of effort to implement, it's hard to measure, it's never "completely" done, and it doesn't bring money to the business. However, while "positive security" is very hard to define, the lack of regular "negative" security is something that companies cash out for in terms of fines, which may even lead to the death of a business. No company is "too small" to think about security. "Oh, we won't be hacked because our application is small and not very popular" is a poor argument. I'm sure the owner of a Winnipeg mattress store thought so too - before the company was forced to pay a criminal who shut down their servers, stopping all the sales.

A Bit Of Background

Do not you have a feeling that developers care more about smooth animations than about data protection? I think the reason might be the gap between the world of product makers

and the world of security people. The gap in their skills, competence, and mindset. I came to work in the area of security and cryptography after leaving the shiny world of mobile development. I was working in a "software boutique" company, creating iOS applications, and NodeJS or Python-driven backends. I managed to put my hands on several dozens of mobile apps, like chats, online shops, medical platforms that process patients' data, apps for controlling smart devices, and so on. We mostly worked with startups and small companies, and we didn't have a separate role of a solution architect or product engineer. I had a chance to be responsible for the whole mobile architecture, the protocols, and API layers between apps, web and backend, the data storage and synchronization, backups and monitoring, etc. Now I am working in a data security company, making software that protects data and prevents leaks, which is designed to be friendlier to developers who are not the from the "security planet", like my ex-colleagues. As I often say, "Cryptography should work everywhere," and so our open source libraries and tools can be found in small mobile apps, as well as in large country-wide infrastructures. Every day I speak with other software companies who care about data security in their products, and I realized that they need help. From these interactions, I noticed that most security people have no experience in developing decidedly usable software. On the other hand, most software developers don't have skills in security analysis and architecture. While I'm staying between two worlds, I feel this problem deeply.

Throwing Hot Potatoes

When I ask developers why they don't implement basic security-sanity features (like protecting user passwords, limiting access to user data, etc.), I often hear the same answer: "The manager didn't tell us to do it, we don't have this task on the board." Imagine having the following conversations with managers again and again: - How things are going with security in your app? - We are totally fine, we have smart and competent developers, they handle everything. - Does your team have security-related tasks? Do they assess the security risks while planning new features? Do you follow the Secure Development Life Cycle? Do you have an internal blue team? Do you do security audits once in a while? - Well... We don't do all of that.. but our QA team does pen-tests! It's a great first step when developers try using pen-tests and security checklists. However, this is a low hanging fruit of "let's build and release quickly, developers will try to do their best, and a week before a release we will go through the OWASP checklist and solve the obvious problems." In the best-case scenario, the team will write down and solve the critical issues, but people usually only solve the easy things and put off the complicated ones to be implemented in the next releases - or, more typically, never. Building secure software is hard. An attacker has to make just a few correct guesses, while a developer has to take care of a number of things from the very beginning. Your system is not secure if you do SSL pinning and store keys in key storage in your app, but use an open MongoDB with default admin password on the backend. Data security works if it covers the system fully, including mobile and web applications, backends, external services, and backups. Solution architects

and security team should care and align feature developers and devops. This approach is efficient when a company is not hiding under "we're fine" umbrella.

Secure Development

It is impossible to learn secure development just from reading tutorials. Most "Building secure chat" tutorials starts with "Let's create a new project" and end up with the "Now we have encrypted data" step. Moreover, they use encryption libraries with APIs expecting developers to choose between symmetric cyphers, their mode *(ECB vs GCM, huh?)*, salt and "nonce, " etc. Most developers just copy-paste cryptographic code snippets without understanding them. Even if data encryption is done correctly, one needs to design the rest of the application flow, use a different encryption for transferring the data, consider the key management procedures, choose appropriate authentication controls, techniques of monitoring and alerting, and so on. As developers, we are always trying to cut a few corners, are not we? I've recently discovered a "codeless" service that "protects" mobile applications you send it to them. So, imagine, you send a compiled .ipa or .apk and code signing credentials to their site, and they "magically" protect your app by integrating encryption, SSL pinning, DLP, and obfuscation. Surprisingly, neither managers nor developers are embarrassed. Close-source system that changes your application flow without providing you with a way to see changes, what could possibly go wrong? The illusion of security is much worse than its absence. Secure Software Development Lifecycle as a methodology that has existed for many years. It is described in detail as MS SDL (deep and solid) and as OWASP S-SDLC (short and modern). The SSDLC distils common sense from the industry experience of building secure software and prescribes techniques which cover most risks in most cases. The SSDLC consists of several steps and accompanies a typical software development approach (including Agile and XP.)

- *Risk evaluation and assessment* - understanding business and technological risks threatening data of and on which the application operates.
- *Building a threat model* - what are the typical threats that the application faces?
- *Security roadmap* defined according to the most possible threats. Typically, a security roadmap contains data minimization, data protection during storage and transmission, access limitation and monitoring.
- *Secure coding* - which libraries and tools to use, where to store keys. Using a good encryption library itself won't make your app secure.
- *Secure operations* - infrastructure-level changes that should be done, including procedures of revoking user sessions, updating certificates, patching libraries.
- *Security verification and testing* are usually done combining manual and automation tools for continuously testing a code base, including its dependencies, for vulnerabilities.
- *Threat response and recovery* - the plan and procedures to conduct upon detecting threats.

The SSDLC is a continuous process that should start after suspecting any vulnerability or detecting an incident. And it's a process, not a single feature to add.

End-to-end Encryption And Marketing

How technically difficult you think it is to make end-to-end encrypted applications? Well, it's a bit tricky, but not impossible. Cryptography helps to reduce the attack surface and risks and prevents attackers and insiders from reading the data. If a system does not know which data it operates on, this data cannot be stolen easily, right? So why do not we have more E2EE data exchange in modern software? If a company does not have access to its data, it cannot analyze it and use it for advertising. Finding the right balance between missed advertising profits and derisking actual ownership of this data is tricky, but doable.

Tomorrow

Secure software development is far from being a popular practice, so we still need reminders to show us the real consequences of data security. Until good software and secure software become synonymous, there will be fines and losses. So, what exactly should we do tomorrow to improve the security of our software? I don't have an easy answer for you. Maybe starting to devote time to security as a sign of professionalism - similarly to the way we focus on writing maintainable, testable, manageable code, not just code "that works"?

Do you want to write for De Programmatica Ipsum? Contributors get paid $150 per article, and are fully credited for their work on the magazine. You also benefit from our writing guide, helping you to find your dpi voice, and editing guidance from Adrian Kosmaczewskiand Graham Lee.
This six issues contained here in Volume One included contributions from eight paid writers, and we will be looking for more to write articles on our upcoming topics. If you would like to get involved, see our guide and fill in the form to let us know what you'll bring!

Issue IV

Programming: Science Or Art?

January 2019

The Art of "The Art of Computer Programming"

Graham Lee

Image: Graham Lee

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

> 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) 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 subvolumes 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, of yak shaving. In the Jargon File, Yak Shaving 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, 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, 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 tool that could "tangle" source into a Pascal program for the computer and "weave" it into a TEX document for a reader. So Web allowed him to make TEX and METAFONT (and the five-volume Computers and Typesetting 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?) will suggest a few more. I do not be-

lieve that I will be stretching Vitsoe's logistics to find shelving for all of the works of art in computing writing.

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.

Why Not Both?

Roland Leth

Image: Clem Onojeghuo on Unsplash

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. 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 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 un-

derstandable 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.

The Creativity of Computing

Carola Nitz

Image: Federica Galli on Unsplash

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 ar-

range 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.

> 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. 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.

A Brief History Of Programming Artists

Adrian Kosmaczewski

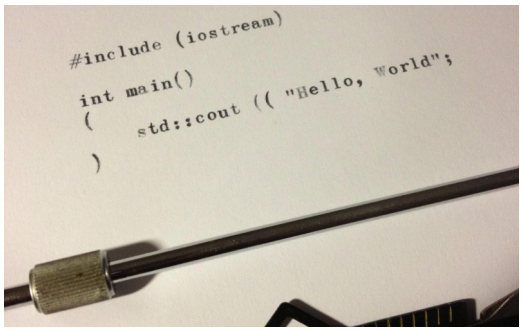

Image: Adrian Kosmaczewski

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, 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" by Richard Stallman, he appears to agree to this definition, from chapter 17, "Words to Avoid (or Use with Care)":

> 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," 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 back in 2005, at a time when Ruby on Rails was becoming a storm that was about to change the face of web development forever.
Yukihiro "Matz" Matsumoto 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" (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" or simply "_why," wrote The Poignant Guide to Ruby, 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 of his work.

Programming As A Self-Fulfilling Prophecy

Tom Murphy wrote a "strange paper" 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 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 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 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, wrote a book about the subject, called "Poetic Computation: Reader" 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:

> 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.

Programming As The Antithesis Of Art

In "Mindfire: Big Ideas for Curious Minds," 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" 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" 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 happiness while making things happen in a computer. 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 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" by Frederick Brooks.
- "The Psychology of Computer Programming" by Gerald M. Weinberg.
- "Hackers and Painters" by Paul Graham.
- "The Art of Computer Programming" by Donald Knuth.
- "Object-Oriented Software Construction" by Bertrand Meyer - for example, on page 878, when he mentions Barry Boehm.
- "A View of 20th and 21st Century Software Engineering", a paper by Barry Boehm from 2007.
- "Dealers of Lightning" by Michael Hiltzik.
- "Revolution in the Valley" by Andy Hertzfeld.
- "The Tao of Programming"
- "Masterminds of Programming" - for example on page 305, where Anders Hejlsberg talks about C#.
- "Design for Hackers" by David Kadavy.
- "The 'Future Book' Is Here, but It's Not What We Expected", article on

Wired from December 2018 by Craig Mod.

What do you think of *De Programmatica Ipsum*? Do you like the magazine? Do you dislike it? Has this collected volume made you want to subscribe, or to avoid us? Are there topics you'd like to see us write about?

We deeply value your feedback. Adrian Kosmaczewskiand Graham Leeare committed to making this magazine a valuable contribution to the society of people interested in software. To do that, we need to know what will make this work for you. We read all messages posted to the website's feedback form, and thank you for taking the time to help us improve our work.

Issue V

Ethics

February 2019

The Current State of Ethics In Tech

Agis Tsaraboulidis

Image: Matthew Henry on Unsplash

Over the course of the last couple of years, more and more press has surfaced about companies and their unethical practices and processes –how they have manipulated or exposed users' privacy or data for their own gain. This is not new, with many of the reported incidents having happened in 2012-2013 and before. In 2018 alone, Facebook's privacy incidents stacked up to 21 and counting. But it would be wrong to target Facebook alone; it is not about a single company but the industry as a whole. Major companies like Uber, Grindr, and Volkswagen were exposed by reporters for their own unethical practices. Though we see article after article published about the act, what I can not find are articles explaining why companies behave that way. In many of the cases after a company is exposed the founder apologizes (sometimes) and moves on. That way stakeholders, employees, and the public are appeased but actual change is not in the plan of the company. Why? Because then it happens again. It is a never-ending cycle. As programmers when we deal with a problem, we break it into smaller pieces so we can understand what is happening and make the situation more easily digestible and find a solution. And that is what I would like to do here as well. Here are the main reasons I assume (based on enough news and press releases to count) companies continue to behave that way and do not respect users and their data.

A False Obsession To Change The World

If you ask a founder what the drive behind their company and the product is, 95% of time they will say they want to change the world. By having this very abstract and broad goal as their north star, they fall into multiple traps without realizing it at the start. By doing so, it removes the ability to think deeply about what it truly takes to achieve that goal. However, the north star is so grand and powerful, it becomes an "at all costs" behavior to reach it.

Leadership: A Focus On Positives And No Plan For The Negatives

One of the most important parts of a company DNA is the ability to adjust the sails after veering off course. It sends a message to the team that we have learned the lesson, we know what caused us to veer off, and we can prevent it from happening again. In a recent interview, Mark Zuckerberg said:

> Facebook was "probably," he admitted, "too focused on just the positives and not focused enough on some of the negatives.

From Fake-news to Russia's interference to US elections, these are now big problems that Facebook did not notice in time and now they

are paying the price. By doing so, they enabled all the negatives to pile up without a plan of action. And even if leadership had it, they did not share. But when patchwork on the ship begins because holes are appearing, there will come a point where the ship begins to sink. Rule of thumb: It is much more efficient and beneficial to assemble a crew to fix the problem rather than patch it up only for another hole to appear. But that is not possible if the crew is not aware of what they are fixing.

Silicon Valley Culture: Profit Over Everything

Silicon Valley is both a magical and mythical place where many of the biggest companies are born and continue to do so. There is no arguing that the drive and spark you get when visiting is like no other. Not to mention the huge valuations, big and unrealistic funding rounds, and culture. But at what cost? Throwing off the economy? Sexism? Depression? Companies growing at a fast pace (regardless of whether or not they profit) need to continue trucking and investors want (and need) a return on their money. So we turn a blind eye. Theranos, anyone? As long as the wheels keep turning, the press keeps coming, and the users are engaged, the funding continues. We have founders who focus only on the positives and a culture that promotes profit over anything no matter what. That is a deadly combination and we can understand that through what happened to many companies. So far we focused on analyzing the problem and the source(s) of it. But what can we do to tackle the actual problem?

1. Legislation Of Tech Companies

There is a huge discussion about legislating tech companies. In many countries, especially in the EU, legislation plays a heavy role in what companies are permitted to do or not. There is less ability to take advantage of users, data exposure (without the proper precautions in place) is not just immoral but illegal, etc. Generally, it is a great idea but in countries like the United States, there is an even more grand problems. The people set to legislate have not the slightest clue how the internet works. Congressional hearings of Facebook or Google's CEOs are clear examples of this. By simply listening to the questions and responses folks from the committee it is not just clear that they do not understand the basics of the Internet's workings, but lack the desire to learn. Young, educated representatives in office are necessary for a new generation of not only people, but technology, and progress. There is nothing that stunts growth more than mindset.

2. Leadership

With or without legislation, tech organizations and their leaders have a choice and responsibility as to whether they behave ethically or not. So what can leaders do to make a company more ethical? To kick things off they need to promote good conduct. Lead by example. If the boss never takes a vacation and stays in the office from 8 am to 10 pm, the team will feel inclined to do so. Use their own behavior to create an environment that promotes more ethical processes within the company. By doing that will inspire their own employees to do the same. In addition to that, leaders need to put unbiased team members with a specific focus on ethics in executive positions; doing

so ensures they can intervene when they feel something is unethical or wrong in order to set things straight - creating checks and balances within the company even when leadership is in the wrong.

3. Kick Off Conversations

As an industry, we need to have more conversations about ethics and how to bring change in our workplaces. This cannot be done by a single person but more likely a group of people. Enough employees that have the ability and privilege to take a stand and make their voices heard to company leadership must do so. So the next time you are in a meet-up/conference or hanging out with coworkers, start a conversation. Social media has proven that enough back and forth dialogue can lead to change. That starts with small conversations.

4. Educate The Public

When an article about an incident around privacy comes out, it will most likely be shared around social media (especially within the tech community) and less likely to be shared at a grand scale by folks outside of this bubble. Why? Many people who do not work in tech most likely do not care about such articles. They do not care because they do not know about how important and valuable their data is to companies, and what the repercussions are of said data being exposed. So we need to educate folks on how to protect themselves and their data on the internet. They need to learn that in many cases they are the product of a company. By educating them the next time a new incident breaks out they are going to be more mindful, understand better what is happening, take action and speak up against the company. We have a long way to go until we fix this problem but if we do not start doing something, this situation will become worse. As Tim Cook said before, "Privacy is a fundamental human right." Companies have a duty toward humans on the other side of the screen, to build a product that they can love and trust.

Primum Non Nocere

Adrian Kosmaczewski

Image: rawpixel on Unsplash

Here is a non-revolutionary idea. One that many have had before, and one that many will have after this article. One that might never become true, because of the forces at play and the strength of some contrarian opinions. Here the author proposes the establishment of the equivalent of the "Hippocratic Oath" for the software industry.

Origins

Software as a craft grew out of the needs for computation of capitalism and cold war. Commercial and military computing drove the development of faster and better computers, through a process that ended up putting computers in our wrists, streets, pockets, pacemakers, and in every planet, moon and asteroid in this corner of the galaxy. Capitalism needed faster computation. Computers, born as a by-product of the Second World War, were perfect for that matter. Computers required software, and this software begat more developers. Some of those developers created faster algorithms, some made simpler programming languages. Some made history, some just made a decent nine to five during forty years. For a long time, software development did not pay any attention whatsoever about ethics. As practitioners, we cared more about compilers, syntax, libraries, frameworks, IDEs, but we did not ask ourselves whether our software was being used for good causes or not. Neither Mark nor Jack did ask themselves that question when they started their famous ventures. They could not care less. Comes to mind that infamous sequence in "The Social Network:"

> A million dollars is not cool. You know what is cool? A billion dollars.

Hubris

Hubris drives the industry. Not ethics. The current state of affairs is a direct consequence of the lack of questioning in the industry. We have created a broken world. Our world is made with software. We are the makers of this new world. We must be held accountable, not only for its wonders, but also, and most importantly, for its flaws. As a profession, we have enjoyed for a long time the privilege of knowledge. We have used the power brought by that knowledge, and we have created one of the youngest professions available to new students these days. We are the new Francmasons, and as such, we must start applying the rules of ethics accordingly. It is time for the software engineering profession to be driven by ethics. And up to a certain point, regulated.

Accountability

It is time for accountability, for the end of the all-powerful End User License Agreement that removes all liability from the creators of software. At first sight, however, and given

the large applicability of software, it might seem that not all branches of software engineering should be covered by regulation. One could argue that an agency making websites or games might not need to be covered by regulation; companies making software for pacemakers and financial trading, most certainly. However, the author of these lines does not believe in the auto regulation of markets, and as such, all branches of software should be regulated by state intervention. Such intervention is required in most, if not all parts of modern economies, and software experts are already part of the ranks of governments and judiciary powers.

Definition

Fortunately, the pitiful state of the software industry these days gives a hint to what behaviours could be considered ethical these days:

- Selling consumer data for advertising purposes is unethical.
- Creating consumer products without respect for privacy and security is unethical.
- Releasing software without accessibility features is unethical.
- Racial profiling of any kind is unethical.
- Collaborating with dictatorships is unethical.
- Using language detrimental for any human group is unethical.
- Promoting or simply allowing the spread of fascism or fascist ideas is unethical.

If your software does any of the things above, you are acting unethically through your software, and you and your team are responsible and must held accountable for it. Each of the developers involved in the creation of such a software (apart from the creators of the libraries and operating systems linked to them,) and in particular those who make a profit out of the sale or promotion of such a software, must be held accountable for breach of ethics. Let us be clear: all of the team members must be held responsible. Responsibility cannot and must not be diluted. Hubris can be lethal:

> Additionally the overconfidence of the engineers and lack of proper due diligence to resolve reported software bugs are highlighted as an extreme case where the engineers' overconfidence in their initial work and failure to believe the end users' claims caused drastic repercussions.

We need software-savvy lawyers defending online harassement victims. Software-savvy politicians approving new legislation for our software-driven world. Society needs software companies to be held accountable for bugs in critical systems. To be responsible for privacy breaches in consumer systems. To proactively prevent security issues in public infrastructure and national security. In general, to be responsible for any kind of gross or lesser negligence related to the software put in production.

A First Step

And all of these measures could start with a very simple first step, taken at the end of computer science, programming, or engineering studies: *an ethical oath*, mandatory to all those working in software systems. The very mechanism that Hippocrates found the key to ensure

a decent standard of ethics in the medical industry, must be put in action in our own industry, and fast. Maybe this will be the nudge that will make many managers in software companies, nowadays only attracted to the industry because of good salaries or eye-catching IPOs, to care about quality, accessibility, inclusion, privacy, and security. Let me be very clear about this: market forces will *not* solve the issues caused by the lack of ethical standards in our industry. Maybe, in a few years time, we are going to witness a new kind of graduation ceremonies, one in which software developers pledge for ethical behaviour.

An Oath For Software Developers

I propose the following one, adapted from the current Hippocratic Oath used in the United States:

> I swear to fulfill, to the best of my ability and judgment, this covenant:
> I will respect the hard-won scientific gains of those software developers and engineers in whose steps I walk, and gladly share such knowledge as is mine with those who are to follow.
> I will apply, for the benefit of society, all measures that are required, avoiding those twin traps of overengineering and releasing untested code.
> I will remember that there is art to programming as well as science, and that warmth, sympathy, and understanding may outweigh the knowledge of the developer or the skills of the sysadmin.
> I will not be ashamed to say "I know not," nor will I fail to call in my colleagues when the skills of another are needed for the proper resolution of a problem.
> I will respect the privacy of my users, for their lives are not disclosed to me that the world may know. Most especially must I tread with care in matters of life and death. If it is given me to solve a problem, all thanks. But it may also be within my power to generate another one; this awesome responsibility must be faced with great humbleness and awareness of my own frailty. Above all, I must not play at God.
> I will remember that I do not merely create a system or implement an algorithm, but I create systems for the highest benefit of society, who will have to use it and who will store their most confidential information within. My responsibility includes these related problems, if I am to solve adequately the problem at hand.
> I will prevent early code optimization whenever I can, for readable code is preferable to fast code.
> I will remember that I remain a member of society, with special obligations to all my fellow human beings, those experts in the field as well as those not initiated or knowledgeable in the matters of code and software.
> If I do not violate this oath, may I enjoy science and art, respected

> while I live and remembered with affection thereafter. May I always act so as to preserve the finest traditions of my calling and may I long experience the joy of solving the most intricate problems I am faced with.

Conclusion

It is said at the beginning of this article that this is not the first time somebody comes up with such an idea; Graham rightly pointed the author to the ACM Code of Ethics which serves basically the same purpose, albeit with a longer text. May any of these ideas find an echo. It is the humble opinion of the author of these lines, that without such mechanism (necessary but not sufficient) we will not be able to start talking about ethics in software engineering properly. As David Heinemeier Hansson said recently:

> Maybe it's time we need an algorithmic oath for programmers: I will program no harm by privacy theft, attention hoarding, radicalization optimization. I will not put engagement metrics above the humans they are extracted from.

What Is To Be Done?

Graham Lee

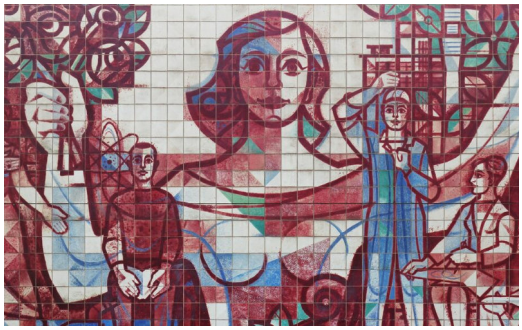

Image: Dmitri Popov on Unsplash

The title of this article has been used and re-used by socialists throughout history. Luke puts it into the mouths of the crowd listening to John the Baptist. In response to their question, John suggests that those who have surplus clothing and food share with those who do not have enough.
From each according to his ability, to each according to his need.
Russians re-used the title frequently during the decline of the Romanov dynasty. Nikolay Gavrilovich Chernyshevsky was a socialist democrat who wrote a novel promoting industrial collective communes under the title. Then Count Lev Nikolayevich Tolstoy took up the mantle in his response. Tolstoy criticised the idle work ethic of both the poor (who should work harder to improve their lot) and the rich (who are in no position to accuse the poor of being lazy). Perhaps the most famous use of the title is Vladimir Ilyich Ulyanov's, who produced an illegal pamphlet under this heading influenced by Chernyshevsky. Ulyanov's "what is to be done?" set out the case for revolution in Russia. It was a work that set the Russian Empire on a course to become the USSR, and Ulyanov to become its first dictator under his pseudonym, Lenin.

Why Do Anything?

The short question is powerful because of the axiom it implies. To wonder *what* is to be done, we assume that *something* must be done. The situation has become untenable, but we want to know what its replacement should be, and how to bring that about, before we act. It is also weak, mealy-mouthed, the question demands a programme of action without hinting that the querent intends to do anything about it. Yes, I agree that this absolutist monarchy governed by an ineffectual Tsar should be replaced. You have given me a specific plan of action to bring that about as I requested. OK, good luck with that Vladimir Ilyich, goodbye!
Hmm. Denouncing the parlous state of current affairs. Demanding a plan of improvement. Declining to actually commit to bringing those improvements about? This sounds like a job for software engineers!

An Empire In Decline

The public seems particularly disinterested in advances in computing at the moment, while either masochistically or necessarily becoming more dependent on it. Facebook is in a perpetual state of apology. Google have been hit with a record fine. The World Economic Forum expect Bitcoin's value to level out at zero. A member of the United States congress is discussing racism in artificial intelligence. A cursory news review of software security shows a range of concerns, from security cameras being hacked to a national early warning network being breached.

Computing "enjoys" a reputation similar to that of genetic modification technology in the 1990s, or nuclear technology in the 1970s. Nobody's really sure what it means, they know it could go drastically wrong. But they also know that it's coming, ready or not, led by the governments and companies who tell us that "progress" is good, whatever that means.

What *Is* To Be Done?

We do need to be part of the discourse and debate around the technologies we use, promote, and inflict upon society. Rather than snarking in our social media bubbles about "the necessary hashtags", technologists need to be seen, heard, and *understood*. And our positions need to be coherent.

And that means we need to do the thing that is missing in the examples given above: we need to *listen*. Policymakers and journalists are reflecting the concerns that are found in society, and if we want to form coherent and impactful responses to those concerns we need to empathise with the people formulating them. Perhaps we need to walk away from particular activities that society can never condone, rather than convincing them that "making the world more open and connected" is good for them despite their misgivings. We have to listen, learn, empathise and understand *more* than we need to educate, inform and correct. While there is value in the thing on the right, we have to come to value the thing on the left more.

Systems Thinking

Any software artefact is the confluence of three complex systems:

1. the software system itself
2. the team of people who design, build and maintain the software
3. the society of people who find their lives, jobs and opportunities modified by the existence of the software system.

Much writing on software engineering focuses on the first system (computer science, software architecture, software implementation paradigms) or the second (development methodologies, team organisation, devops). Not so much the third! The NATO 1968 conference on Software Engineering - the event that marked the birth of the field, had a brief discussion on user input into design. Quotes range from:

[J.D.] Babcock [designer of timesharing computing systems]: In our experience the users are very brilliant people, especially if they are your customers and depend on you for their livelihood. We find that every design phase we go through we base strictly on the users' reactions to the previous system. The users are the people who do our design, once we get started.

to:

[Brian] Randell [IBM, leader of the conference's working group on design]: Be careful that the design team will not have to spend all its time fending off users.

Empathy

We can argue over the specific time at which empathy in software design entered the software engineering body of knowledge. Barry Boehm's 1999 paper "Escaping the Software Tar Pit" introduces the modified golden rule to software engineering:

Do unto others as you would have them do unto you – if you were like them.

But maybe we're doing Mike Cooley's 1982 "Architect or Bee?" or Don Koberg's 1983

"The Universal Traveler" a disservice. Either way, it is only now that many teams are coming to appreciate that the product owner may not have all of the answers about what makes a good product. The fields of User Experience and Design Thinking encourage us to see how people respond to our ideas, to take their criticisms and suggestions on board, and to design the system that *they* will benefit from. Fundamentally we need to remember that that third system exists. It is society, and we are its designers, its architects, and must bear in mind that we are a minority of its members.

Issue VI

Diversity & Inclusion

March 2019

Why I Want People To Not Treat Me Differently

Susanna Riccardi

Image: Annie Spratt on Unsplash

Before I start, I should give a little introduction about myself. I hope this helps you understand my point of view. I'm a 22 years old woman from Italy with a Bachelor's degree in Informatics who is currently working in Sweden as a Full Stack Developer. I started programming about four years ago when I enrolled in University.

An Attempt To Improve Diversity

Recently I was on Twitter and I read a tweet from UIKonf, a conference for iOS developers taking place in Berlin, announcing that for this year's edition they are going to have an all-female lineup of speakers. Here is what they say on their website:

> We want to do our part in supporting women in our industry and showcase some of the great women speakers out there. This is our way of raising awareness for the diversity problem while at the same time a celebration of all the women who follow their passion, muster up the courage to go on stage, and hold great talks despite the headwind they're facing in our industry.

What the conference decided to do is to select speakers according to their gender, instead of basing their decision on meritocracy. Knowledge and experience are put in second place to help a cause. In the conference organiser's view, trying to bring more women into tech could potentially reduce the gap between men and women in the field.

The Reactions

Judging from the replies to that tweet, there were generally two kinds of reactions. Some people were supportive towards the conference decision, saying that this will help with inclusion and diversity of underrepresented groups. Others were criticising it, claiming it is now excluding everyone else. There were a couple of replies that caught my attention, because they were expressing what I felt the moment I saw the conference announcement. Both of them were wondering if the conference decision didn't actually go against the aim of equality and inclusion, expressing that a lineup made up on skills or topics would have made more sense.

My Opinion

Here is why I disagree with this approach of having these targeted experiences for women. I personally find it diminishing to be invited to a conference, hired at a company, and generally being treated differently because of my gender. This is something I cannot control, unlike my education, my experience or

my attitude towards other people. If what everybody is looking for is to be considered the same as anyone else, shouldn't we look at these values that any human being can actually influence? I don't want to be prioritised because I am a woman. I want to be selected based on my knowledge, what I am able to do, and what I can contribute to what I want to take part in.

My Personal Experience In The Field

What is the point of spending resources in inviting more women in tech, if the environment is not sustainable enough to retain them in the long run? I have read countless stories about women and people from other underrepresented groups having a hard time at their workplace. I have not been in the field for long, but in my short time therein I have had some bad encounters with men who wanted me to feel inferior because of my gender. I have had to prove myself many times to show people that I had the right to be where I was, whereas men at the same skill level didn't have to: everyone simply assumed that they knew what they were doing. During my time in university, I experienced classmates pointing out that I was getting good grades because of my appearance. Others told me I was passing my courses because my boyfriend, who was attending the same program, was helping me. On the starting day of my first tech internship, the CEO told everybody that he was happy to see a woman join the team. I questioned myself all the time when I was there - did they merely hire me because I am a woman?

The Environment

What happened to me is nothing compared to what other women have to go through. Various researches have been done regarding the differences between men and women in tech: how much they earn, how much their opinions are valued, and how long it takes to progress in their careers. The work environment is not optimal and welcoming, and we have data to prove it. Women are leaving the field more than men do. So what do we want to achieve by just inviting minorities to work in tech if they will end up leaving because of the hostile environment? Let's imagine for a moment that I am working at a company where I experience harassment by my colleagues because of my gender. In order to help improve the situation, the company prioritises their hirings based on gender. Why would the men that are harassing me stop doing it, just because they are surrounded by women? Wouldn't they think that those women got their job without having to prove themselves? Wouldn't they think that the newly hired women don't deserve to be where they are because of this? I know I am just assuming their behaviour, but I find it hard to believe that such men will magically become nicer people because they are surrounded by more women.

Equality

This is why I don't feel like supporting this movement. I think it is going against what we are fighting for. I hope you don't misunderstand me: there is so much we can do to make it better for everybody, and we have to try different approaches. But we need to be careful with what we are doing: some actions can do more harm than good. My wish is that everybody, regardless of gender, colour of the skin, sexuality, and any other characteristics, can feel at home in the tech field, without having to feel excluded because of something

they cannot control, but also without feeling merely included for the same reason. Being a developer is one of the best jobs in the world, and there is space for everyone. We need people with different backgrounds, experiences and ideas. They can bring so much more than just improving the diversity statistics of a company. I don't want us to mainly focus on bringing minorities into the field, but on making the environment so good that they will never want to leave. I don't want us to focus on diversity either. That is not the primary goal. I want us to focus on equality. If we can reach equality, diversity will come by itself.

A Suggestion

I have had huge discussions with many people, and some of them told me that this is a longterm goal. We will not be able to see any concrete improvements soon, but our grandkids will. If this is really the case, why are we not focusing on fixing the problem at the root? I believe shifting our focus to promoting engineering to kids in schools could give amazing results, much faster and much more effectively than simply targeting minorities. We need to show our children and teenagers that tech is not only for men. Everybody has the same chances of becoming a software engineer as anybody else, and nobody is privileged.
I didn't get into tech until I enrolled in university because I was always told that this field is only for men. When I was in middle school, my parents begged me to not join the high school I wanted because I would have been the only girl there. They were scared I could feel alone and discriminated. That's why I was reluctant to join my university. But I am so glad I did, because I have met awesome people, of any gender, and I have worked with them in a field that has a lot to offer.
If girls grew up learning computer science, they would see its beauty, and would consider it when making their decision when thinking about their future. If boys grew up studying engineering alongside women, they would finally see their potential and would treat them as equals. I strongly believe in this solution: we could achieve our goal, diversity in tech. And for once, we could stop begging women to join the field because we just want to improve the ratio.

Conclusion

There are numerous things we can do to improve the situation for everybody. We need to dare trying different strategies, but also be critical of the current ones. Everything has pros and cons. I believe having opportunities targeted for underrepresented groups that exclude other ones does more harm than good. We should make it so who people that are working in tech want to stay. Let's concentrate on shaping young minds, on showing that this field is not just for men. It's for everybody.

Sources

The Systems Holding Back Women In Tech, Tracey Welson-Rossman, May 10, 2018, Forbes
Women in Tech: What's the Real Status?, Laura Garnett, Mar 21, 2016, Inc.

Hiring Diversity (Beta Version)

Julia Cacciapuoti

Image: Raj Eiamworakul on Unsplash

The only constant is change... *and the hiring process.* The way companies design their hiring process has always kept my attention. But during the last couple of years, with the increasing discussions and researches about inclusion and diversity, this became even more special. We are walking through a new and super interesting paradigm, but still in beta version. Life habits changed, technology evolved and evolves every day, access to trainings and tools to develop our skills became more and more available, but there is something that believe it or not maintained its basics: the hiring processes.

The ~~Un~~conscious Biases

It's incredible to meet colleagues, no matter located where, and hear about the same tips, tools and assumptions when it comes to source and screen talent. The more common approach to analyze profiles is full of biases. Starting with a global perspective, the University you studied in and how long it took, says a lot about your capabilities. More than that, depending on where you studied, you will probably have more chances of being considered than other people that decided a different path. Of course there is no room for people that didn't find their path in the academic world. It's not a good sign, so it's better to not waste time and move on to another profile. If you jumped from job to job in a relatively short period of time, you are a job hopper and you don't deserve my time as recruiter even to ask you why or how you made these decisions. But, it's not all negative. Some countries are more advanced with some aspects. Didn't look for photos in the resumes. Recruiters don't ask about age or personal life. These companies use to have Diversity and Inclusion practices that also apply to their hiring process, paying special attention to the gender neutral language in the job postings, the pics in their website, the training for interviewers; just to mention a couple of initiatives. But the *best practices* about sourcing and selecting the *best talent*, remained the same. Moving to a local view of this situation and double clicking in Argentina, the situation is even worse. We still ask for personal data. If you are a female, not a minor number of companies still find super relevant to know about your personal and family plans. We pay attention to the photos in your resume. We analyze your date of birth and do judgments for free. We have a decent but still small and understaffed IT market, so making assumptions about people trajectory becomes really easy. We, as recruiters and "employers" use one of our best attributes to predict who you are based on your profile. We use our arrogance to quickly arrive to conclusions. We strongly believe that our market has less than 10 respectful product companies, with a reputation earned about their products but also the high bar in the hiring process. We

also assume that if you went to public universities, you have a plus. Although we are a bit more familiar with the idea of not necessarily finished your formal studies or take almost 10 years to get a bachelor degree, this keeps our attention anyway. If you are not studying, or if you withdrew, or if you are studying at an University that social trends say that it's not at the expected level, we assume to know a lot about who you are. So we automatically know how could you potentially perform in our hiring process and of course in the role, our culture and company. And please don't tell me you are more than 40 years old and you still want to code. What happened in your career? Then it becomes an obsession to find out a problematic relationship with a former boss or any related typical problem. If you are not considering become a Manager and you just want to code, we need to address it because this could tell a lot about your ambition, intrinsic motivation and willingness to grow. Another interesting point about doing hiring in Argentina is related to the super mentioned and manipulated concept of Digital Transformation. So, in the future (?) all companies will become a technology company. We have been repeating this for a long time. However, we didn't change our assumptions. If you are working at a bank, or at a traditional telecommunications company, you will probably not match with our positions. If you didn't work for at least one of the respected tech companies, something is wrong with you. Why did you not try to take your career to the next level? But honestly, we don't have the answer because we actually do not ask. Contacting and considering these profiles, it's not worth it. We need to optimize and prioritize our time.

The Beta Version

Perhaps give a *beta version* is too much. Perhaps we are even at an earlier stage. Each time I hear someone saying that the profile was archived and rejected because there was not enough evidence of being a good match with the role, my ears hurt. Each time I identify a bias around the educational and professional background, the idea of challenging the status quo and really invest in training, gain more and more strength. Then we intend to work on the Impostor Syndrome, that curiously we are also responsible for generating. What are we trying to say when we are looking for the *top talent*? What does a top talent look like? Who did define those parameters? Once again, there are not answers to these questions. But, hold on. One of the definitions of beta version is *"an early version of a program or application that contains most of the major features, but is not yet complete. Sometimes these versions are released only to a select group of people, or to the general public, for testing and feedback. This is the second major stage of development following the alpha version, and comes before the release candidate"*. So, perhaps beta version is a correct term to define the hiring processes these days. Absolutely aligned with the idea of *"not yet complete"* but still not aware that the *testing and feedback* stages started a long time ago and we are missing the mark. If we still can't see that a human being is much more than a LinkedIn profile, we are in trouble. If the professional experience and the way people make decisions don't motivate us enough to reach out and stop assuming to start asking, we are in trouble. If we are still waiting for people to say politically accepted answers, we are in trouble and we won't be doing any other thing than contributing to the Impostor Syndrome. But more than that, we will be losing

our so valuable and obsessive focus on hiring the *best talent*. The real one; that that we don't know.

To Alienate Or Not To Alienate, That Is The Question

Dear recruiters, dear colleagues: we have a lot to do with this. We are driving the processes. We are the proxy between people (or profiles, as you prefer to call them) and opportunities (or openings). We have a critical role and a great power; and great power comes with great responsibility. Define if we want to be aligned or alienated must be our first one job on a daily basis. We can't allow us to keep on doing the same things anymore. We can't seat down and just see how the world changes, read about Inclusion and Diversity, feel pride about the Inclusion practices in our companies but not realize that we will be on the opposite sidewalk if we don't clean our lens. Please do not repeat what Digital Transformation means if you still dismiss people with +10 years of coding experience at companies whose core is not technology (yet). While paying attention to the gender neutral language in our job postings is super important, it becomes cosmetic when we don't encourage each other to do a better job. A responsible recruiting job can't be done without love and passion. An HR role can't be successful without challenging the business and thinking out of the box. Let's move out of beta version and revisit every single day why we do what we do, how we do it and embrace the unique opportunity we have between hands to impact the industries, companies, markets and hundreds of people we are in touch with every day.

Moving Out Of Beta Version

Long paths start with a first step. We don't need to wait until the *perfect* moment. As long as you wait to have all in place to start, the later you will do it. So what could you do starting today?

- Network > LinkedIn: not everybody is in LinkedIn.
- What are you looking for?: ask yourself this question every single day.
- What really answer that question?: which information included in the profile really gives you insights? If you are looking for an experienced Mobile Engineer with ability to mentor other people, does the University where she /he studied tell you something about that? I don't think so...
- The summary: if someone took some time to introduce her/himself in the summary, take your time to read it. Is there anything that caught your attention?
- Too empty profile = become curious: not everyone loves to include full and detailed experience in their profiles. This is not the preferred language; engineers write code so that's the best cover letter. I know! We don't read code. But that's not an excuse to only read LinkedIn profiles. I challenge you to give the next 10 "too empty" LinkedIn profiles a chance. Ask questions to the empty fields and then complete them from the candidate's answers.
- The iceberg model: we don't know what we don't know. I like to think in the IT industry as an iceberg. There are a lot of people showing what they do but in a language that we don't understand.

There are a couple of things you can do about this, as for example build a strong partnership with the engineering teams to strategically work on sourcing, the pitching message and approach or reinforce the referral program, but the most important one is embrace a growth mindset. Don't tell yourself that you don't need to understand "technical aspects". I would like to share this article from Stack Overflow, that provides useful techniques to expand your searches and be able to see more angles of the iceberg.

- Reduce assumptions: challenge yourself to not complete unknown information with your assumptions. Develop a genuine interest in knowing people and their stories. Always keep in mind that there is much more than a profile behind a profile.

We have the unique opportunity of revalue the concept of candidate experience, trying each candidate as we like to be treated as customers. Details make a huge difference and challenging old and unprovable beliefs will generate the inflation point in the way that companies recruit people. All this is also about inclusion and diversity. Let's move about of beta version. All we need is passion, love and conviction.

Sheree Atcheson On Diversity And Inclusion

Adrian Kosmaczewski & Sheree Atcheson

Image: Sheree Atcheson

An exclusive interview with Sheree Atcheson, an award-winning Diversity and Inclusion leader. She spends her time helping organisations create inclusive environments which embrace people of all walks of life. She currently works at Deloitte UK as the Tech Respect & Inclusion Manager, at Women Who Code as a Board-Appointed Global Ambassador, and as a Contributor at Forbes.

What are the most pressing problems regarding Diversity & Inclusion in the tech industry right now?

We are not having teams which represent the societies we live in. Technology is innovative, fluid and creative, and to fully meet the needs of all, we must have teams which replicate the society which are using them. Our organisations should spend the time understanding what our people need and how we can provide that to them, allowing them to grow and flourish in their work and levels.

You have been very active in the promotion of D&I; based in your experience, what are the most successful concrete actions undertaken to solve the problem? (By yourself or by others)

Relatable role models - role models are crucial. It is a big ask to ask someone underrepresented and/or marginalised to also have the emotional energy to push through and be the first. When we see people similar to us succeeding, it is empowering and clearly shows it is possible to succeed. Sponsorship - sponsorship is incredibly powerful in the advancement of underrepresented people. Sponsoring someone means advocating for their career, providing constructive feedback to aid growth and lending visibility to their work. By doing this, allies are helping actively make a difference in someone's career. Engagement of allies - we must have the help of the majority in making the industry better for the minority. Diversity in technology is good for all and through allies realising their power in shaping a positive impact, we can create actionable changes to aid more inclusive environments.

Could the effectiveness of those measures be improved, or do we need a "paradigm shift" to tackle the problem?

Both. Yes, we can measure the number of visible leaders or increase on retention of underrepresented groups however, this will not shift overnight - this is going to take ongoing work to continue to change the ratios in relation to all aspects of diversity. Organisations embracing diversity and fostering environments of inclusion are having healthier bottom lines because they are listening to more inputs and voices, reducing risk in solutions and creating

the solutions for the many.

Have you seen an improvement in the situation in the past few years, or has the situation simply changed without signs of improvements?

Yes, we can see differences because people are having these kinds of conversations and organisations are aware that changes need to be made. The more awareness we have, the better - however, we need actions post-listening.

Regarding those improvements, what are the "metrics" (if there are any) that can be used to gauge them?

The percentage of underrepresented people in industry, especially in leadership positions. Whilst we need representation at all levels, including junior roles, we need to ensure we are providing environments where underrepresented people can flourish and become leaders. Also, to spend the time in understanding your attraction, retention and exit data - to form strategies around that, instead of a blanket approach.

Colophon

De Programmatica Ipsum - Volume One
ISBN: 978-3-906926-25-4

This volume of De Programmatica Ipsum was produced by exporting the website articles in Markdown or HTML format and converting them into TEX with the pandoc tool. The collected TEX articles were then processed with XELATEX to generate the volume.

The Fell Types are digitally reproduced by Igino Marini, and used with gratitude to him.