The Great Idea of Async Work

Adrian Kosmaczewski

2022-10-14

As I mentioned last week[1], I've been in this industry for exactly 25 years. I started my journey as a software developer on Monday, October 6th, 1997. I've had the opportunity of sitting down and writing code for a living for a quarter of a century!

Put it differently, I can say I've been active in this industry for 34% of its existence, if we take 1950 as the "year zero" of computing, or 48% if you consider January 1st 1970, or "epoch,"[2] as the initial point.

One third or one half. In any case, it's quite some time.

Think about this: for how long have there been carpenters, blacksmiths, farmers, fishermen, or merchants, in the world? Well, now think how long 34% of that time would be. Many hundreds of years, certainly, at least for some of those professions. That should give you an idea of how young is our craft.

That brings me to the following idea: we're at the very beginnings of the computer industry. We're still pioneers in this discovery. It is all still very much a "far west." We're still figuring out how to do things with computers. This is all extremely new from the point of view of history.

This is the reason why there are so many new things all time; new frameworks, new languages, new methodologies, new technologies every six months. This rhythm happens because as a species, we're just learning how to deal with computers. We're learning how to build software, day by day. And we're slowly getting better at it every day (well, I'd like to think that!) even if it sometimes doesn't seem so.

This is also why some technologies stay with us, and become what we label as "legacy," like COBOL, for example. I hear some people chuckling in the back, please stay with me.

Now we're hearing about "quantum computing" and whatnot, and at some point we're going to leave the Von Neumann architecture behind[3]. So far, all of the computers in our world have used the Von Neumann architecture. From the Apple Watch in your wrist to the controllers on board of the Voyager probes. From those beloved Raspberry Pis to those IBM mainframes running COBOL code every time you swipe a credit card.

[1] /blog/running-akosma-software/
[2] https://en.wikipedia.org/wiki/Epoch_(computing)
[3] https://deprogrammaticaipsum.com/a-farewell-to-the-von-neumann-architecture/

Great Ideas

In the past 72 years, we have found some ideas that worked better than others; the Von Neumann architecture was clearly one of them.

COBOL is another. It just works, and it keeps delivering, even if the syntax looks unpalatable to Rust and JavaScript developers today.

Agile is a great idea. It's like we realized that software engineering deals with a substance radically different to concrete, wood, or metal, and that people could actually perform better and create a better thing if we started valuing the things on the left more.

The Model-View-Architecture (MVC) is another great idea, even though React seemed to be replacing it during the past decade, and I'm not entirely convinced it was a good idea, to be honest. But in its own right, MVC was a great idea, and it helped build so many different products in its many incarnations.

In programming languages, we have lots of very recent ideas that have been proven to be fantastic: automatic memory management, for example, either through a garbage collector like in Smalltalk, Java, or .NET, or more recently at compile time, using automatic resource counting like in the case of Objective-C or Swift, or using a borrow checker[4] like Rust's. This has freed software developers from the burden of having to keep track of all those objects on the heap in their heads, and then risking to run out of memory if they didn't, and crashing if they tried to access things they shouldn't be accessing.

Generics is another great idea, allowing developers to make sure that a bag of red marbles only contains red marbles, and neither green marbles nor red tokens. Generics are so much a part of our world that even Go has adopted them lately[5]. They have also brought many benefits: together with enums, Generics brought the idea of `Optional` types to the mainstream, which are making Sir Hoare's billion mistake[6] slowly become a thing of the past. And a similar idea can be used to replace exceptions, thanks to a generic `Result` type. These are all great ideas.

Composition instead of inheritance; here's another great idea that finally has grown in us; when I started programming in object-oriented languages such as Java, inheritance was all the rage. These days we know that it's better to build software in small pieces, each well defined and tested on its own, which we use like Lego bricks to build bigger and better software, incrementally. We even reached the point in which neither Go nor Rust have inheritance[7]. Think about that: two of the hottest programming languages today don't feature inheritance, at all.

Type inference[8] is another great idea; it allows your code to have the benefits of a strong type system, but without the verbosity. Your code looks like a scripting language, but everything is strongly typed, and as such it can be optimized and

[4]https://doc.rust-lang.org/beta/rust-by-example/scope/borrow.html

[5]https://go.dev/blog/intro-generics

[6]https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[7]https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/

[8]https://deprogrammaticaipsum.com/the-truce-of-type-inference/

verified in so many interesting ways. And our IDEs can pick up type information as we write our code, and they have become substantially better with time.

Functional programming is now a feature of most languages these days: you can have immutable values in your code, and then `map()`, `filter()` and `reduce()` your data in PHP, C++, C#, Java, Go, Swift, Kotlin, Rust… and of course JavaScript, which was the most misunderstood programming language[9] 20 years ago, and now it's one of the most widely used. Those functional constructs simplify our code, because thanks mapping, filtering and reducing chunks of data, we got rid of lots of `for` and `while` loops, which made code easier to read and maintain.

Unit testing[10] is another great idea; many modern languages integrate a simple unit testing library in the language, so that you could start doing TDD from day one: for example D, Go[11], or Rust (You could. That doesn't mean that people actually do, but they could. They should. I hope you would.) Each one of your components can have its suite of tests, and then you can tell your CI/CD pipeline to execute them for you automatically, and you'll get an e-mail as soon as the build breaks, to make sure that your code is as strong as possible, all the time.

We have also stopped using the `goto` statement, following Dijkstra's timeless advice[12]. It took us 50 years, but we did it, finally. Even if James Coplien says[13] that polymorphism is `goto` on steroids!

Distributed source code management is a great idea; after many different failed attempts at storing source code (Microsoft SourceSafe, anyone?) we went from centralized to distributed, and now Git[14] reigns on top of the category. It has become a standard, and that's a good thing. I personally haven't had to use any other SCM tool than Git for the past 14 years–I still had customers using Subversion and Mercurial around 2008, but they moved to Git ever since.

Containers are another great idea, and Kubernetes[15] is also here to stay; it offers a very flexible environment to run apps. Thanks to containers we don't hear anymore the famous "it works in my machine" excuse; we literally ship your machine. A simple `docker run` or `podman run` will make code, including all dependencies, to download and run in just one operation. I cannot stress how revolutionary and unprecedented this is; I actually expect this pattern to spread to other environments, in particular smartphones and desktop computing.

We have learnt how to deal with legacy code, and that's a great idea; we have lots of good advice and literature that help us as engineers to encapsulate, refactor, test, and talk to legacy code every day. This is knowledge that has only been recently gathered and filtered, and we now have that knowledge at our disposal, and that's a great idea.

[9] https://www.crockford.com/javascript/javascript.html
[10] https://deprogrammaticaipsum.com/kent-beck/
[11] /blog/d-or-what-go-may-have-been/
[12] https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf
[13] https://deprogrammaticaipsum.com/james-coplien/
[14] /blog/git-for-non-technical-readers/
[15] /blog/kubernetes-for-non-technical-readers/

Not so great ideas

Pay attention to the fact that I have not mentioned Machine Learning in this list; I do not believe it to be one of the greatest trends of our time, and I believe that there will be yet another AI winter[16] coming up soon.

The biggest issue I have with ML is that we're not eager to see our own biases face-to-face. And ML does precisely that: it slaps our faces with our own biases. Hence, we are appalled at our own racism, our own exclusionary practices; we do not want to tackle those issues, and as such, the practice and uses of ML will get stuck.

And don't get me started on blockchain, web3, NFTs or any of that. At the current stage, it's an unregulated scam of gargantuan proportions, and even worse, a colossal ecological disaster in terms of negligible delivered value by megawatt of energy spent.

Asynchronous Work

And probably the greatest of all great ideas, Open Source[17], has stuck and is here to stay.

Why do I think it is the greatest of ideas? Because it is based around the notion of collaboration and sharing. What we're discovering through software is a new form of social organization. Which is kind of ironic, when you think about it. As practitioners, we're not particularly known for our social skills, are we?

Yet, I am seeing a huge new trend, one that is closely associated to the Open Source movement, and that will have incredible impact in the years to come.

We have just went through the third summer with a pandemic around us. The virus is still there, but you get the idea. As we get out of our homes, some of us might miss those times working from home. It certainly had some advantages.

Many businesses have been very quick to scrap the "work from home" policies they were forced to setup during the pandemic. For many reasons; in some cases, rightfully so, for there are jobs that cannot be done remotely.

But there are a lot of jobs that *can* be done from anywhere in the world nowadays, among which software engineering; why don't those jobs stay remote?

There are (at least) two reasons for this. The first one is hard to change, has very deep roots, and is called **trust**. Some company cultures simply do not have trust built-in, and that's a very difficult thing to change. So I won't talk about this, because there's not a lot you can do to change trust (unless you're the CEO, that is.)

But there is another factor that prevents companies from going into full remote work mode, and this is a factor that each one of you can contribute to change.

And that's **working asynchronously**.

[16]https://deprogrammaticaipsum.com/open-letter-to-a-future-ai/
[17]https://deprogrammaticaipsum.com/open-always-wins/

Working asynchronously is a different thing than working remote, although one is eminently compatible with the other, of course.

When then pandemic started, many companies jumped to full remote mode, of course, following the instructions of the government. As days started passing by, however, some companies realized that things were actually working very well. They were more productive, and even morale had improved.

At the same time, we heard the horror stories of countless people around the world struggling with remote work. It became clear to some of us that, without knowing it, there was something we were doing right, that others weren't.

And that was something people now call asynchronous work. Inadvertently, many companies had been working asynchronously the whole time. They just didn't know it was called that.

Writing

Working asynchronously works best when you follow some basic requirements.

The most important thing about asynchronous work is that you must write, write, and write. Decisions must be written down, meeting notes must be written down, specifications must be written down.

If you don't like to write, well, maybe asynchronous work is not for you.

Teams working asynchronously usually adopt the concept of RFCs[18], the same ones that were used to create the various networking protocols we use today, or KEPs[19] (Kubernetes Enhancement Proposals.)

Writing is very important for asynchronous work. When decisions are written, they can be agreed upon; when using GitLab or GitHub to manage those RFCs, people can comment, send pull requests, even raise issues about those documents. For each big project, make sure that you write down requirements, features, and ideas, and do that openly; use Git repositories for that. They work really well.

Having ideas in writing allows people to work on them at any time, following their geographical location or sleeping patterns.

Some people work better in the evening, some in the middle of the night, some are morning people. This means that a nice side effect of asynchronous work is that you are more inclusive in terms of neuropsychological patterns.

Talent is not something that works exclusively from 9 to 5. That's a myth. We all have our rhythms.

Since you're writing things down, another side effect is that you need less meetings. Asynchronous makes it possible to redefine even the standup meeting[20].

You can choose to only have meetings in case of deeply complicated issues that require discussion and brainstorming. Have somebody to always write down notes during them, and make sure they end precisely when the timer stops.

[18]https://www.ietf.org/standards/rfcs/

[19]https://www.cncf.io/blog/2021/04/12/enhancing-the-kubernetes-enhancements-process/

[20]/blog/the-various-styles-of-standup-meetings/

Meetings are a very expensive thing in companies, and you should only call for one as a last resort.

When you work asynchronously, everyone is fully responsible of their tasks, outputs, and deliverables. Asynchronous work makes collaboration is a core value in every company.

For engineers, tools like Visual Studio Code's "Live Share"[21] extension are very useful to do pair programming over the network; no need to be next to each other, just collaborate on code with the same tool. You can also use it to write long documents in Asciidoc or Markdown.

Async is great for deep work; engineers appreciate the possibility of focusing their attention in one single issue at a time, without interruptions of any kind, in whichever environment they prefer. Unfortunately, many companies are keen on the concept of open spaces[22], which is one of the worst things that have happened to software engineers.

Open spaces and software engineers don't work well together. You can't have quality software in open spaces. Async work opens the possibility to engineers to work on problems at their own rhythm, in silence, and without interruptions. I cannot stress how important this point is, particularly now that every company is a software company[23].

Another nice side effect of asynchronous working is that stress levels are much lower. Async work is incompatible with micromanagement, and if people need time to go to the doctor, walk the dog, help their kids with homework, or anything else, they just have to notify their teams, leave, do their thing, and pick up where they left later on. Of course you will have deadlines, but managers should instead count on collaboration to help everyone reach their objectives.

Async begets trust and fidelity. Attrition rates go down, because people enjoy both flexibility and being trusted to do their jobs.

What about the Office?

But what happens to our beautiful office? Well, maybe it's time to downscale it, for example by not renewing the lease of some of the space you're renting. This will bring some economies, of course, while still having a nice office for people to come and work into if they prefer.

Let us be very clear: remote work is not for everyone. Some people truly need and enjoy the daily contact with their peers, and having a physical space for them is paramount. The important thing about async work is that your office becomes yet another remote location for your activities; one that happens to have your colleagues in person, and that's a great bonus.

So, if you can, keep your office; use it for team gatherings, apéros, hackdays, company-wide announcements, and for all those moments (and minds) where collaboration requires physical presence.

[21]https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsliveshare
[22]/blog/open-spaces/
[23]https://www.forbes.com/sites/techonomy/2011/11/30/now-every-company-is-a-software-company/

Tools

What tools can you use for async work? Markdown[24], Asciidoctor[25], and Antora[26] can be used to create documentation websites, put together with GitLab[27]. Merge Requests, just like GitHub[28] Pull Requests, are great async enablers.

Confluence[29] is a fantastic tool for async collaboration, and it even allows many team members to collaborate in real time on a same page.

Use a chat system like Slack[30], Mattermost[31], or Rocket Chat[32] for live interaction. But even though chat and Zoom[33] are great for live communication, Discourse[34] works better for longer discussions, questions, and actual async brainstorming sessions.

These tools are 100% cross-platform and browser-based; because it's 2022 and people should use the operating system that they prefer. Many software engineers use Linux laptops; designers prefer MacBooks, and management types are more often than not Windows users. They must all be able to interact with one another. Web-based software is compatible with all of these operating systems and more (BSD, anyone?) so people can work as comfortable as possible.

For async management purposes think about using a web-based ERP, CMS, password management tools, a CRM, a bug tracker like Jira[35], and other tools. Use SaaS software whenever possible, particularly if your organization is small, or you don't have a dedicated IT team.

These tools are an enabler to remote and async writing processes; and in turn, writing is an enabler of async work.

Books

If you are interested in async and remote work, I can recommend two great books about it, both written in 2013:

- Remote[36] by Jason Fried and David Heinemeier Hansson; it tells the story of Basecamp, a SaaS company at the origin of the Ruby on Rails web framework.
- The Year without Pants[37] by Scott Berkun; describing the story of how WordPress works, following the author's incursion in the company for a whole year.

[24]https://en.wikipedia.org/wiki/Markdown
[25]https://asciidoctor.org/
[26]https://antora.org/
[27]https://about.gitlab.com/
[28]https://github.com/
[29]https://en.wikipedia.org/wiki/Confluence_(software)
[30]https://slack.com/
[31]https://mattermost.com/
[32]https://www.rocket.chat/
[33]https://zoom.us/
[34]https://www.discourse.org/
[35]https://en.wikipedia.org/wiki/Jira_(software)
[36]https://basecamp.com/books/remote
[37]https://scottberkun.com/yearwithoutpants/

I also recommend you subscribe to The Async Newsletter[38] prepared by Doist[39], the same team that brought us Todoist[40] and Twist[41]. They know a thing or two about async work, too.

Conclusion

Writing is an enabler or asynchronous work.

Open Source is an enabler of asynchronous work.

Asynchronous work is an enabler of remote work.

Async is the actual breakthrough, the actual secret sauce of companies like Basecamp, WordPress, GitLab, and many others. This is the factor that made them successful when working remotely.

There is no point in working remote for the sake of it; working asynchronously makes your company more flexible, resiliant, stable, and agile than ever, and it enables you to go remote if you need to.

I believe that async work is the next big thing in management. Well, not only me, but many others think so[42].

And thus, async work will become the next great idea brought forward by the computer industry.

[38]https://async.twist.com/
[39]https://doist.com/
[40]https://todoist.com/
[41]https://twist.com/
[42]https://async.twist.com/