Post-software engineering

Challenges for the 21st century

Jakub Šebek, 2022 (1st public draft)

Hey, Alexa! Would you write me a short rsync script?
— Do you have time to teach me? (1)

The computer of the future is different from the computer of 2022. Perhaps not so much on the hardware side, but in its software — greatly. It doesn't stand in one's way, it doesn't hinder the user, and it doesn't act unexpectedly. Promises as old as time remain unfulfilled and society is left crippled. At a point ever more critical, we have to take a few — or a few more — steps back, think very carefully, and continue in the direction of Human-Machine symbiosis that we really want.

Note: It is to appear obvious the author concerns himself merely with the near, foreseeable future, roughly corresponding to his expected lifetime — the 21st century. Or similarly, until the term "computer" loses all meaning as the natural course of advancement renders this concept itself ridiculous.

Prologue

The first scenario suggests a future where man stays relatively dominant in his relationship with the computer. In this case, the computer's physical form doesn't undergo a drastic change — we have a keyboard, a mouse and a screen — or components strongly reminiscent thereof. However, the assistance received is indispensable, and the interface is perfectly ergonomic and fault-free. The user hardly even thinks about the existence of the computer, being completely immersed and undisturbed in whatever pursuit their curiosity, boredom or job needs them.

The computer of the future in the second stage loses the well-known peripherals — all that is left is the fundamental screen with an audio input/output interface or, if the user require silence, a stenotype of sorts. A direct brain-link interface is also not unthinkable anymore. The commonality of these input methods is that they mediate the communication in natural language (or any form of communication that is the most comfortable to the human — we assume this to remain natural language for Homo Sapiens in the time span we are considering).

Ultimately, the computer poses no barrier to the user whose thinking capacity becomes virtually unlimited. The computer continuously learns to adjust to its master — to understand him better and to serve him better. At the same time, the master develops an attachment to the computer. Inevitably, the computer becomes, at least on its outside, a person — for no one can understand and comfort a person better than another person.

Ordinary people will be left with a lot of time to anthropomorphize and question these robots, but engineers must not sleep, as their train is headed into a rather different future — and it doesn't take stops.

Regardless of whether these are two separate stages, two timelines, or parts of a mix, they underline several common goals that the field will have to overcome before they can materialise. We hitherto observe these Achilles' heels of software engineering:

1. Reliability and stability

By far the most sought after feature of a computer as of today — for it to "work". In fact, the ongoing incompetence has gone so far that it doesn't strike anyone as remotely odd nowadays when a computer doesn't do what it is expected to. People have fully accustomed themselves to the fact that "Computers don't work". There are few feelings more dreading than trying to complete a task and finding our previous approach dysfunctional — that likely because of the mistake of some human. The damage caused every day as a result of software error is unquantifiable.

2. Security

Just looking at the heaps of information that the computer infrastructure accommodates, all the vital aspects of our lives it controls and the amount of harm at stake must entice us to place security of computers as a top priority. Similarly to computers not functioning properly, people are getting used to systems getting circumvented, accounts getting "hacked" and networks "infected". The response to such announcements I never hear is "How is that even possible?", and acceptance of reality is not the reason for that. With the protected property being fully digital, we finally have everything necessary to mitigate such offences completely, yet we are still failing.

3. Automation

The majority of end-user software developed today is still poorly equipped with automation interfaces that would allow the human to, one day, decide: "Would you do this for me?". Many automation solutions and frameworks exist, but they are limited to domains — the automation of user software requires industry-wide attention. One is for certain — more and more work is going to be carried out automatically, by robots. In order for the transition to take place as smoothly as possible, we shall prepare the appropriate software infrastructure for the future — and sit back for the Great Handover. It is one of the last inventions we'll have to make, after all.

The remainder of the work is dedicated entirely to these three questions. In a manner of gradual exploration, the author attempts to prove the paper's core thesis: **Not only does the industry converge in issues, but their solutions converge in philosophy — one idea, one vision, and one radical paradigm shift.**

Part One: How did we get here?

In the beginning, there was lambda; then, Church said: "Let there be computation!" — and there was computation.

The computer is first conceptualised by mathematicians — men of utmost elegance, purity, and aversion against the slightest of imperfections. The tape, the combinator, the state graph — is whence the machine comes by its soul.[1]

Computer science is born

Like explorers of new worlds, computer scientists are the biologists and zoologists in the wonderful archipelago of computation unveiled to us by mathematics. They became the half-step between mathematicians and engineers — they took the abstract machines, they disassembled them, and carefully studied their behaviour. They even programmed some of the first computers, but their overly methodical approach would soon discourage anyone who wanted to make any money off of the new miracle.

Software engineering is born

One truly must not mistakenly downplay the contribution of computer science to his life with a computer. Every action one takes in his work — each mouse stroke, every key press — as soon as these step from the physical to the digital ether, thousands of computer science papers sparkle at every stop and turn taken before pixels appear on the screen.

However, scientists are strictly never the cohort that brings innovation to the general public — We have engineers for that. Naturally so, software engineers storm the world to turn academic papers into reality.

The software engineer

It didn't take long for governments, universities and companies to start adopting computers. Responsible for this boom were the first software engineers; despite the ever-growing requirements put on them and the tight limits of contemporary hardware, they never failed to deliver — They laid foundations for the silicon age by creating the first programming languages, operating systems and development practices.

These engineers of the early computing era were also the first humans to see their program misbehave — they found the first bugs and crashes, and they were the first to hold their heads in eternal dread of their coding errors. They faced unprecedented problems and pioneered solutions. Their actions were affected by many factors, such as the economy (their bosses), their peers, but primarily, their human nature. A distinct culture vividly emerged — the same culture that would, decades later, bring their entire discipline to a close.

1 As we'll see, pure mathematics remains a major source of fuel for the field.

Demigods

Mechanical engineers are held back by the laws of physics, astronomers cannot fight the movement of celestial bodies, and doctors subordinate themselves to the might of biology. One thing that computers allowed their creators to do than ever before was prototyping new ideas and solutions on a whim — making only their imaginations the limit. If architects could modify material properties, astronomers change orbital trajectories and doctors design the body (we're getting there!), great responsibility would lie upon them — the same one we gave to software engineers.

They turned the big cumbersome calculators into something pleasant and useful — first for themselves, and in turn for everyone. They transformed the bits and bytes of silicon to beautiful user interfaces and tools that every user or programmer of the previous generation had only dreamt of. But will they reach the utopia they have been so determinately working towards? Are they ever going to be full? Total power doesn't couple well with raw human greed and the constant desire for more satisfaction. Let's see how our new demigods coped.

> *The engineer evolved to be pragmatic, make quick decisions and focus only on the necessary. This gave him a great ability to swiftly adapt to new demands and overcome challenges on the way.*

Most people, including most software engineers, would happily agree with this description; perhaps it would even be celebrated. But what if the phrasing is slightly altered?

> *The engineer evolved to be impulsive, reckless and tunnel-visioned. This makes him prone to hasty solutions and leaving heaps of duct tape behind himself.*

Suddenly, an entirely new shadow is cast. We cannot dismiss the features highlighted here as they are a direct, valid — but most importantly — real and observable implementation of the first description. The fact that we managed to show this only by specialising the original, innocent — or even jubilant — portrayal reveals the danger implied by such a seemingly normal expectation for software engineers — even the well-meaning ones.

This exercise certainly isn't presented for the author's cynicism or as a dead end of the argument. We can start pulling strings and watch the history of software engineering paint itself as it unravels our three-question predicament.

Nature or nurture?

We can come up with several hypotheses as to the impulsive nature of the engineer. Fans of politics wouldn't be afraid to go to great depths explaining the effects of the exploitative nature of the system most of these engineers worked in. On the other hand, those of us oriented more at psychology and the true origin of human

action look for the answer deep in the individuals' heads.

The rapid onset of computing was definitely not the first nor the last time man faced a dopamine problem. In fact, we can observe this in the adoption of any new exciting technology by humans. Sometimes it is expected, or even desired, and other times it is unexpected — but in most of these cases, depending on the novelty's extent, the result of such reckless and irresponsible action can be disastrous.

We might also ask whether the culture of software engineering would end up taking a different path if we moved its epicentre from the USA to Southern Europe, The Soviet Union, Eastern Asia or The Middle East. These places around the world are known for their distinctive social schemes, habits, work ethic, forms of leadership and ways children are raised. But would any of those be really able to steer away the innate human motives?

Humans are driven by incentives; they act upon them subconsciously in good faith, but what for is good faith when they aren't hunting and gathering?

Strategy

It is hard to talk about any kind of strategy when taking a closer look at the trends in software engineering. By strategy, we mean systematic, long-term thinking — evaluation of criteria, problems and solutions. In many cases, the lack thereof is becoming a conscious choice.

Strategy takes time and intelligence; strategy must be taught and trained; mainly, strategy has to be called for and valued — incentivised. Neither the customer nor the manager of the age do so, however. Rapid development is being cherished in favour of thorough thinking; that is, apart from a few areas.

Exploring the layers of societal action leading here allows us to hypothesise the flawed management of modern hierarchies, the recent invention of consumerism and customer-oriented free-market capitalism, the overall rise in wealth that made people needy and insatiable; all the way down to the impulsive primal mindset. But today we are not toppling the house of cards of industrial societies — only the one of software engineering; and it is frankly difficult to imagine software engineering as a whole becoming a thing weren't it for these "evil" standards.

A similar lack of strategic incentives can be seen in other circuses of the world. In some cities, people are getting tired of their roads being repeatedly torn apart and paved again every so often; watching seemingly pointless construction take place, or having governments debate ridiculous affairs only for their non-existent or short-lived outcomes. The growing sentiment of futile and exhausting team meetings is especially ironic, as such sessions are likely seen as the vital triumph of the company's strategy by its higher executives. (2)

Anyone who has ever played chess was imbued with the sudden realisation of the

importance of strategy in the real world — every move needs careful planning, as every move can have catastrophic consequences, and it is only in the hands of the player for which side. With the general lack of strategy and the lack of self-reflection today, we cannot expect something like *Reliability and stability* to show as a defining aim.

Accessibility

As the growing army of unstrategic software engineers started shaping the virtual world to their ultimate liking, one of their primary focuses was their own comfort. Via models of abstraction, they started creating indispensable tools, such as high-level programming languages, libraries and interfaces. This way, they lowered the barrier of entry to the field significantly. You suddenly didn't need a degree to use a computer, and a new generation of engineers started eagerly jumping onboard.

This new generation no longer had degrees, and started slurping on the fruits of abstraction left by the the fathers. However, as we will see, abstraction has one fatal flaw, and these fell absolute victims to it. With the henceforth lacking background, more errors and oversights begun to pile up, and worse — these innocent practices obviously started to spread, including to the latter generations, as this was just the beginning.

It's not like there were no software engineers with degrees after this point, or that some essential knowledge had been lost — that would go against the only accelerating rate of discovery and true innovation that followed. The surging wave was simply too strong; layers of abstractions continued towering up and lower stories were quietly being forgotten. The software engineer of the new era adopted a modern mental model and never looked back.

Moore's curse

When Gordon Moore pointed out in 1965 that every year, double the amount of transistors was able to fit in the same area than previously, he knew what kind of amazing advancement lied ahead of computing. What he forgot was that humans make disobedient gods; and that giving them more resources could turn out like giving the newts more guns.

Even faster than the astronomical growth of transistor counts grew lines of code, job positions, and levels of indirection. Computers were suddenly an order of magnitude faster than a few years ago — and did almost twice as much! Sales went up, developers had their legs on the desk, and everyone was happy.

Not all, but a bigger portion of the current ills of software can be categorically attributed to Moore's famous law, or, less anecdotally — the whopping power, speed and capacity increase that is accompanying computational hardware to this day. This is an undisputable fact, as the development of modern software and the newly adopted practises are largely built on the premise of performant machinery — sometimes even before such machinery is commercially

available[2], and simply couldn't take place with more limited resources. We can observe growing general sentiment (4) against this status quo in particular; indeed, many call for turning back the time for computer hardware.

These causes — by which software tends to expand to fill the available memory — and the roles of both the fallacious brains of the customer and the developer in the conundrum, have been famously discussed for decades (3).

Collapse of complex software

We have learnt how irresponsibly relying on abstraction devised by giants upon whose shoulders we stand — which Moore's law has allowed us to do relentlessly — gives rise to complex software. Of course, today, deep inside, every engineer knows this (partly because we have been entering a certain hiatus in performance growth); but it is far too late. Software has become unmaintainable and by no means suitable for our vision of the future; it will inevitably collapse and bury all fantasies of contemporary software engineering once and for all.

Complex societal organization notoriously leads to collapse (5). Assuming this perspective, we can arrive at surprising and full-fledged conclusions (6) about what the "software society" is going through, or where it is headed. Juxtaposing with the former Roman Empire allows us to draw interesting parallels about its rise, the golden age, and possible subsequent fall. Notably, we certainly observe the Roman imperium fall, but we watch it live on and its spirit flourish in its successors — whomst eventually surpass it. This can fill us with the right amount of faith, courage and care needed for taking the first steps as a post-software society.

A brief software prehistory

1985 *— Bjarne Stroustrup creates the C++ programming language, marking a definitive turning point of indifference to complexity.*

1990's *— Object Oriented Programming becomes the dominant paradigm of trends and development. It grabs focus of thorough study, and gives birth to entire ecosystems, cultures and principles such as SOLID. (7)*

1995 *— JavaScript, the archetypal pillar of modern software, is born in 10 days (8). This language would later prove a major accelerationist force in the collapse.*

2009 *— Node.js and its npm package manager emerge, the latter of which normalises "dependency hell", introducing the era of the bloated web.*

2013 *— Release of Electron, a HTML/CSS/JavaScript-based platform for deploying desktop applications. Gaining traction later into the decade, this technology creeps on the thinnest ice so far, provoking an uprising of doubt in both userbases and developers. (9)*

~2040 *— Collapse of software as we know it*

2 Such became a traditional form of advertising "next-generation" videogames.

Part two: Post-software

What do the Sapients do about their precarious situation? The philosophy of software has been solved in the 20th century. The key ideas were left here for them to embrace — and it is only up to them if they listen or not.

The era we are entering might as well go by "Software Renaissance", as it clearly includes by and large returning to the foundations, learning from questions that have been answered in the past and persistently ignored, practises that went out of fashion and many ideas simply swept under the rug. This recurring arrogant assumption of our infallibility and total superiority over the previous generations is what has ultimately led us here, and an attitude we will have to quit soon. (10)

Schools of thought come and go, though, never in a linear fashion; in fact, many teachings tend to coexist and intertwine at the same time — influencing each other and evolving in parallel. The same is true for us — by no means will we have to do absolute guesswork and fabricate our projections from thin air, not only do we incorporate ideas of the past, but ideas and software of the future that has already started to arrive. This part studies many of these timeless and key programming paradigms, concepts and doctrines.

A tour of the brain

Because we are still humans, starting at the very heart of human cognition is essential for designing better tools of software engineering and getting a broader view of its problems; as we've seen, almost every societal dilemma can be traced to the basics of what makes us human. So far, development paradigms have been designed "by engineers for engineers", but that might change; while these engineers exhibit a remarkable ability to help themselves, they are naturally susceptible to blunders — ones that affect generations. Directly or indirectly investing cognitive psychologists and other scientists-outsiders deep inside the engineer's mind — or in the actions of the industry as a whole — could prove increasingly valuable. (11)

The human brain has evolved into a magnificent form which allows us to reason about non-trivial dynamic systems of ever-growing size in a myriad of ways. It achieves this by matching patterns — anything that matches a simple enough pattern, be it entirely imaginary, is going to be picked up by the brain. Indeed, this concept is so general that the brain can be viewed as nothing but a looping pattern-recognition machine. It is also incredibly effective at learning new patterns and making connections in them; perhaps even too good, as this ability of vertical pattern

stacking makes possibly the last piece of the secret called abstract reasoning.

Naturally, we strive to design a system that is as much in line with the engineer's pattern set as possible. This is the reason we use English names and familiar syntax in computer programs and why we invent paradigms that we can understand better; the reason we talk about certain classes of graphs as "trees" and arbitrary memory locations as "objects" carrying "types".[3] None of those concepts actually translate to the hardware we use to run the programs, they are but servants to the biggest humanist project in the world called software engineering.

Extending the mind

When we design programming languages, interfaces and abstractions, we are designing extensions of our existing minds. This is where many programmers leave their mind self-reflection, but the ocean is much larger — the imaginary world is only where it starts. The engineer's mind isn't confined in the tiny space of the cranium; rather, it lives on the outside. It extends to physical world, and includes all that is useful to him — the pen and paper, the book, the whiteboard, and ultimately the computer, is what comprises the mind. (13)

It is important to advise that not taking the opportunity and care of extending the mind, or extending it in misguided ways, doesn't lead to a mere inconvenience or a slight lag in our ability — it severely cripples the psyche like a malnourished infant. The bigger our mind is, the more surface area surrounds it that allows its natural growth. Each such extension has the potential to conflict with the existing realm, possibly becoming a burden, but we can replace such parts trivially, and getting the extension right endows us with superhuman prospects. A good bit of attention in self-improvement of every engineer should go towards taking care of, growing and always refining the healthy super-mind.

Adopting this understanding turns the mind from a static and limited product nearing obsolescence to a wonderful organism that moves, evolves and adapts; but mainly — one that can engineer itself, with nothing standing in the way. In its fullest, we define the global society as one functioning organism, and each single individuals' mind as spanning the entire society. Everyone possesses a skull with a divine embryo in it; bring them together, and we can touch the stars.

Communication

It is arguable whether humans are hindered by their limited ability to write or speak on their own, but this bottleneck becomes further noticeable in communication with other humans. In most cases nowadays, we are nothing alone, and the rising rhetorical competition urges everyone in the field to develop better communication skills. This includes engineers working in conjunction and articulating problems to higher management as well as general audiences.

3 Malbolge (12) makes a curious example of a programming language designed completely against the usual human pattern set.

We likely find ourselves in a time of the widest disconnection between the user and the developer. We can notice developers undertake more and more seemingly unsolicited tangents as is characteristically observed in trends of user interface design, where killing functionality and responsiveness has become the norm in favour of large text, buttons and similarly questionable choices. Creating an unbiased methodology for studying the real feedback and effects on users is incredibly difficult in this case, and thus we can only draw from limited samples, but we can generally speculate about a mixed user experience to say the least.

Another piece of evidence for this gap is the common disparity between software intended for developers and for regular users; e.g. we can see a general tendency of developers to more systematic, reliable and simple tools, such as sleek desktop environments and terminal applications, and then their bulky and complex counterparts used in offices around the corner. We shall add that such software, which finds immense support in the developer community, was once the kind of software everyone used. Improvements in direct and more accurate communication between consumers and producers of programs are a needed upcoming necessity.

Abstraction

In the theme of abstract reasoning, abstraction is a term engineers use to describe methods of simplification of various systems — bringing them from their physical, fuzzy, unpredictable and complex implementations to elegant models that are viewed as a whole and expose only their necessary parts. This powerful concept allows us to take a component of great intricacy, reducing it to but a square on a whiteboard.

By definition, abstraction simplifies ideas — and with every deal of abstraction, some information is lost. This is usually a conscious choice and even a welcome one; a bit of unessential knowledge about the system is simply disregarded in the process. While, from a theoretical standpoint, this leads to underutilization of the system, the advantages abstraction brings decisively outweigh such concern.

Because every caste in the software engineering pyramid can follow this same process for anything they deem unnecessarily complex for their purpose, a certain danger starts lurking. We can see that layering just enough of these piecemeal sacrifices gets us into a real pickle — too much information has been neglected, after all. The most noticeable toll usually taken here is in performance, but in fact, it has the possibility of affecting everything; each lazy engineer in the chain may be tempted to cut corners, pushing the symmetrical square further from reality, and sinking everybody else. A great challenge for developers of the tomorrow will be realising whether all problems in software engineering cannot be solved with one less level of indirection, after all. (14)

Dependencies

What was once seen as a biggest generosity and privilege, that is, using computer code of an acquaintance, has very much become commonplace under the baton of the realm of internet. Of course, this is a phenomenal achievement, and it is now ever more powerful in the reign of open-source software — suddenly, everyone from around the world can share smaller or bigger pieces of code that make their work invaluably more convenient. To developers, having the ability to simply grab a piece of code and ship a product using it the next day makes package managers feel like magical oracles.

In the recent times, we have seen the community particularly embrace such complete and sophisticated solutions to sharing code — no longer is it a snippet you snatched from a forum, or a script someone with the same problem wrote for themselves and decided to publish; dependencies are a consumer commodity with heavy machinery supporting it. Don't even think of marketing a programming language today without its own package infrastructure and a central repository where programmers can publish their libraries. Such packages (each project tends to use an original, quirky name instead) range vastly in function and purpose — we find everything from trivial convenience procedures to highly sophisticated solutions. There are cases where even end-users are forced to use these language-specific installers to acquire consumer software.

But clearly, the developers of dependencies want to use their favourite dependencies as well. This is a key feature of these technologies — dependencies are recursive. They form uncontrolled magnificent trees that end up spanning hundreds or thousands of nodes and monstrous amounts of disk space, with likely a significant portion of them being abandoned, obsolete, superfluous, vulnerable, or all at once. In short, modern dependency chains make one of the purest and most delicate embodiments of cumulative abstraction error.

One of the most common issues an engineer of this age encounters are indeed the ones concerning dependencies. There are many ways to share code and many ways to ensure the process go as smooth as possible, such as careful curation and integration of foreign code for particular purposes. However, the immediate convenience of such efforts falls behind and what prevails is the deadliest combination of granular, recursive packages that remain in reach like cartons in a supermarket. This issue hasn't gone unnoticed; there exist admirable efforts at providing better solutions targeting stability, reliability and reproducibility. (15)

A dependency is something that used to exist on a much smaller scale and with special status, even a term that used to be pejorative, but became completely normalised; in other words reliance — an idea that must instinctively itch just hearing, being the arch enemy of all engineering.

Having seen how abstraction can be done irresponsibly and cause wildfires, we can move on to the bits and pieces that will actually survive the upcoming great filter.

Types

A recurring pattern in human reasoning contemplates objects — little blobs of information and their associated action that ultimately compose into computation. For our own sanity, we give these blobs defined structures for later access, we make them interact with each other, and we even put them inside each other. Finally, our programs tend to reuse many of these blob structures and their morphisms — an idea so natural to us that the development of type theory outran the first electronic computer. (16)

Types are a natural, but also an extremely powerful and robust way to constrain grammars, and the more constrained our programs are, the less computation will be required for their analysis — and the more thorough analysis we can perform on it. In mathematical terms, they are able to effortlessly sieve all possible programs into their minuscule subset of desirable variants — type systems give programs meaning.

The usage of types has become so ubiquitous in coding that their users rarely view them as anything special. While they are given certain superficial attention, the common understanding doesn't go deeper than basic types, perhaps generic types, methods and interfaces. The field is much more extensive and riddled with hidden romance and usefulness that mathematics has been unwinding for a greater part of the last century. (18) As impressively as rudimentary type theory reduces the possible error space, we can go much, much further, and unlike most contemporary programming languages, whose fossilised type systems are kept frozen in time, seemingly awaiting excavation for their display in museums.

Functions

While types are the easiest way for humans to conceptualise and categorise patterns of data, an equally natural extension must be that of modification and interaction of the same data. When we isolate only the items we are working with at the moment, conceiving a universal box that takes some information and returns a useful result, we uncover functions — the very atoms of computation. These again stem from mathematics, and are found in some form in all programming languages.

What makes functions powerful is that they can be thought of individually, as Lego blocks that can be arranged in ever-higher levels of abstraction without spiralling out of control. Having the ability to ponder about just one piece of the puzzle at a time and nothing else gives us the incredible prowess of coming closer to understanding the system in question as a whole. But this is, from another perspective, a rule — a rule that imposes constraints on the programmer, but one that clearly brings great benefits if sincerely fulfilled.

The modern imperative paradigm of programming emerged very early on, and managed to maintain its dominance effortlessly into the present. In this way, engineers could seemingly get the best of many worlds — declaring and mutating variables wherever they felt, writing down arbitrary commands in sequences, and under some circumstances, putting some of them into separate functions. But the cardinal sin henceforth committed against functions went by overlooked. They became no more the crux of all play, stepping down from the central role of any computer program to merely serve as another tool among many. It is way too easy to dismiss unappreciated constraints as limiting, blindly cherishing freedom instead, and thusly all the advantages and guarantees of reasoning with then-functions soon evaporated.

Immutability

Mutation is what we term the altering of information held in memory. This altering takes place for clear reasons, such as memory being limited, but also the common possible benefits of changing topologies of data in favour of faster processing on modern hardware. Perhaps not so coincidentally, mutation also happens to come very intuitive to humans — as the world they live in is subject to constant change, and if something, it is the lack thereof that is questioned. Due to reasons discussed, this notion naturally ended up projected into programming language and software design.

While mutation can therefore be beneficial to both humans and computing hardware in terms of efficiency, there is a big catch. The circumstances in which it be favourable to each differ extensively. In countless cases, a mutating program makes the most sense to a human, but proves very inefficient, and conversely, efficient programs are often unintelligible to humans. The impossibility of mutation, however, is a different story. Provided strong enough type and function-based grammars, systematically constraining mutability can lead existing program analysis to more decisive and complete conclusions as well as allowing optimisation that might at last reintroduce mutation, but now in a provably even more efficient manner. The engineer's willingness to take time and adjust his primal pattern set but slightly to see farther in the end finally makes the last piece of the puzzle for post-software to win on all fronts.

All such systematic constraints reduce entropy to give order, order begets meaning, and meaning enables knowledge. To humans, knowledge furnishes reason and carefully refines and enlightens the mind, deepening its understanding and impeding it from wandering far off the path, and thus thwarting the genesis of complexity. To machines, knowledge equally provides necessary working data, improving automatic analysis, optimisation, error detection, and soon enough as well — reason.

Proof and error

Every engineer has had their compiler coldly spit an error message on behalf of one of these three constraint systems in their face; and we know that engineers rarely take such insults lightly. However, we necessarily emphasize the infinite usefulness of compiler errors, as they embody everything their type, function and immutability systems stand for, and should be most appreciated. In fact, such a pre-emptive error is just another way the compiler tells us that it had hereby proven the program's incorrectness, or even vulnerability against adversaries.

Indeed, we can observe languages with especially lax typing rules suffer from the exact correctness and some security errors that even simple type systems disqualify with ease. But basic type checking is just the beginning. A proof is an unquestionable argument towards the truthfulness of a proposition — define the proposition as "this program is correct", and conquer. Of course, it will still take us a while to reach that point, but that is not to say that proving software is futile — many modern analysis techniques from computer science and type theory are entering wide use. (17)

A proof method that involves direct formulaic proofs known as formal verification has gained traction in the last decades, as it has been successfully used for proving the full correctness of microkernels (20) and compilers. (22) However, despite the technique's merits, it can suffer errors from incorrect specification, and is incredibly time-consuming. Hybrid approaches for direct inclusion of formal verification in programming languages are being brought up (19), but rapidly changing requirements and incentive issues make general adoption of formal verification techniques improbable.

Therefore, we expect more elaborate static verification systems to be embedded directly into languages, possibly carrying over more formal primitives, but posing minimal annoyance to the programmer and empowering them — turning ever more runtime errors to compile time errors. Many of these systems already exist, but dispersed on papers or in isolation (21), where their full potential in conjunction cannot be witnessed. As humans start acknowledging more of their limitations, machine analysis of computer code will become one of the defining features of post-software.

Rehomogenisation

Trying to automate arbitrary computer actions, despite sounding tautological, can prove difficult or even impossible with today's software architecture. User-facing programs are mostly designed with just one interface in mind — manual input from the user and graphical output. The more advanced ones offer their own automation features, except that at the end, those are also confined to a human interface.

In the beginnings, there was nothing available but a keyboard and a text terminal; thus, the only way a user could work with the computer was textual input

and output. The bandwidth was scarce and computers slow, so the interface had to be necessarily simple and possessing only what was necessary. External storage was equally limited, so neither the worked data nor the actual programs could take up vast amounts of space. This lead to the realisation that doing more with the computer would have to involve piecemeal interoperation of multiple powerful programs — ones whose compatibility was paramount. Because of these severe constraints and the desire to maximise functionality, substantially more actual code of the system was exposed to the user, who could then build up this code for almost any use case necessary.

All these barricades were eventually eradicated by the rapid progress of computing hardware, and we witnessed the rise of software in its current form. Space and speed no longer being an issue, humongous pieces of software, like web browsers, naturally arose — and because they could function completely on their own, their usable surface area ratio shrunk drastically. Their internals are comprised by otherwise unintelligible interfaces that will never see the light of day, so not only is the user incapacitated in the usage of the program, but even more so are other programs.

If we expect to do more with computers, and computers to do more for us, we must promote reuse and wide applicability of software. The only way in which computer subsystems can cooperate is if they are able to talk to each other. Many of the aforementioned language-specific package repositories come close to this reality, building layers above existing software that permit its automation. The isolation to a single language doesn't pose an issue if we are confident enough in building this ecosystem into a monopoly, but these building blocks still suffer from big variability in interfaces and their function, and the inherent necessity for programming these complex layers, which might not even be realisable in the first place. We can automate some, but can we automate the automation?

Programming interfaces for web services already enjoy a great deal of homogeny in their architecture. They can be used to reliably query weather forecast, mail, and real-time data. They demonstrate on a network scale what kind of language all post-software could use to talk among itself once — and perhaps finally enable Alexa to write you a short script to sync your class materials.

Conclusions

Different realities make it hard to label software engineering as living through either its infancy, puberty or elderly years. Regardless, the industry is still very much turbulent and will require care to relocate to calm waters — despite the promises made about the future we take for granted, the hypnotising wonder of computation has been driving us further and further from our destination, rather than towards it. There is no one or everyone to blame for this course of events, but it can be viewed as a necessary step in development, bringing countless lessons — prerequisites for a reboot and a new flourishing age.

The future lies by and large in acknowledging our insufficiencies, rethinking our long-term incentives, and gravitating towards simpler and more tractable solutions. The essential problems we face today are not too greater in complexity than countless problems we've already solved in the past with slower computers, or no computers at all. Yet we have voluntarily handed our lives over to machines that no single person understands anymore, machines we are unable to learn about in their entirety or repair ourselves, machines riddled with fatal human error — ones that have kept society afloat so far, but whose use is unthinkable in their current state for the brave new world.

Humans find themselves at a supremely important point in history from many aspects. The future is very promising, and I believe that we are, as a society, capable of overcoming all the roadblocks — but only if we can connect, cooperate, communicate and build upon each other's contribution; if we don't stay ignorant, short-sighted and selfish in our actions; and if we stop looking at the world in its present, but also in 50, 100 and 1000 years, as that is not only the world our children will live in, but where the ultimate fate of humanity strives. Living in the most important century yet, you and I are the most important people to live thus far — we are the giants.

Improve this draft by sending feedback via e-mail.

References

Compelled readers are strongly encouraged to follow some of these sources in particular as they lead to deep rabbit holes of wonderful knowledge and perspectives that the we often touch on only insultingly briefly. In fact, this essay was written only to shine light onto these existing, much more developed works and ideas, as well as to challenge contemporary tendencies, incite reflection and response.

(1) Derbinsky, L. [Lex Fridman]. (2018, March 20). MIT AGI: Cognitive Architecture [Video]. YouTube. https://www.youtube.com/watch?v=bfO4EkoGh40

Part One: How did we get here?

(2) Cutler, J. (2022, July 14). TBM 30/52: Why Don't We Have a Strategy? The Beautiful Mess. https://cutlefish.substack.com/p/tbm-3052-why-do-we-have-no-strategy

(3) Niklaus Wirth. 1995. A Plea for Lean Software. Computer 28, 2 (February 1995), 64–68. https://doi.org/10.1109/2.348001

(4) Handmade Manifesto. (2016). Handmade Network. https://handmade.network/manifesto

(5) Tainter, J. (1988). The Collapse of Complex Societies. Cambridge University Press.

(6) Lawson, N. (2022, June 9). The collapse of complex software. Read the Tea Leaves. https://nolanlawson.com/2022/06/09/the-collapse-of-complex-software/

(7) Martin, R. C. (2000). Design Principles and Design Patterns. https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

(8) Eich, T. [Tech Talk]. (2018, August 6). A Brief History of JavaScript [Video]. YouTube. https://www.youtube.com/watch?v=GxouWy-ZE80

(9) Beyer, C. (2017, November 8). Electron is Cancer. Medium - Commit Log. https://medium.com/commitlog/electron-is-cancer-b066108e6c32

Part two: Post-software

(10) Blow, J. [DevGAMM]. (2019, July 10). Preventing the Collapse of Civilization [Video]. YouTube. https://www.youtube.com/watch?v=ZSRHeXYDLko

(11) Norman, D. A. (1986). Cognitive engineering. User centered system design, 31, 61. https://www.researchgate.net/profile/Donald-Norman-3/publication/235616560_Cognitive_Engineering

(12) Olmstead, B. (1998). Malbolge. http://www.lscheffer.com/malbolge_spec.html

(13) Clark, A., & Chalmers, D. (1998). The Extended Mind. Analysis, 58(1), 7–19. https://web-archive.southampton.ac.uk/cogprints.org/320/1/extended.html

(14) Oram, A., & Wilson, G. (2007). Another Level of Indirection. In Beautiful Code: Leading Programmers Explain How They Think (1st ed., pp. 279–291). O'Reilly Media.

(15) Dolstra, E., de Jonge, M. and Visser, E. "Nix: A Safe and Policy-Free System for Software Deployment." In Damon, L. (Ed.), 18th Large Installation System Administration Conference (LISA '04), pages 79–92, Atlanta, Georgia, USA. USENIX, November 2004. https://nixos.org/~eelco/pubs/nspfssd-lisa2004-final.pdf

(16) Church, A. (1940). A formulation of the simple theory of types. Journal of Symbolic Logic, 5, 56 – 68.

(17) Klabnik, S., & Nichols, C. (2018). The Rust Programming Language (1st Edition). No Starch Press.

(18) Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4), 471–523. https://doi.org/10.1145/6041.6042

(19) Hansen, B. (2022). Magmide; Provably correct software is possible and necessary. GitHub. https://github.com/magmide/magmide/blob/main/README.md

(20) Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., & Winwood, S. (2010). seL4. Communications of the ACM, 53(6), 107–115. https://doi.org/10.1145/1743546.1743574

(21) NASA. (2014). IKOS: Static analyzer for C/C++ based on the theory of Abstract Interpretation. GitHub. https://github.com/NASA-SW-VnV/ikos

(22) Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, *52*(7), 107–115. https://doi.org/10.1145/1538788.1538814