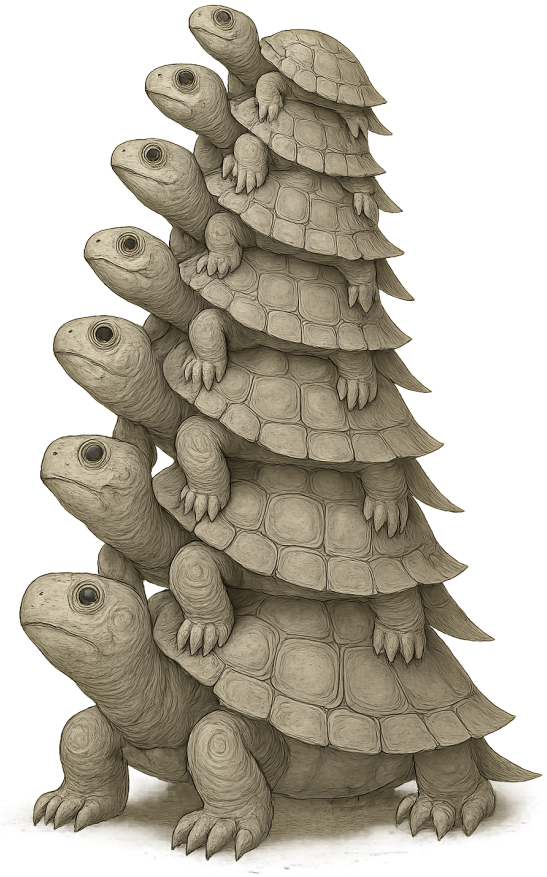
> ABSTRACTION

Software Development or the Art of Abstraction Crafting



Vincent Lextrait
vincent.lextrait@gmail.com
December 2025
V1.2

100% HUMAN CREATED

(except illustrations)

Introduction

The thoughts in this document are the result of 50 years programming almost on a daily basis at professional level, from computers with ferrite core memory to the ones containing ARM processors, from assembler, Pascal, C and Ada to the most modern object-oriented programming languages we know today. I used to be a hobbyist and a tinkerer, but I have left this long ago to concentrate purely on real-life practical applications. The text below deals only with that aspect. I believe any person involved from afar or closely into software development should be familiar with the concepts presented in this paper. The ideas should be at times obvious, or possibly eye-opening and even shocking for many.

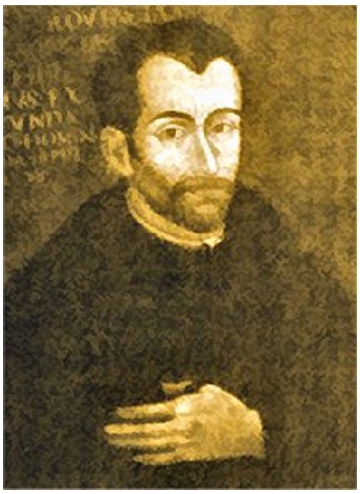
We humans use many abstractions. Our entire view of the world is based on them, starting from our language. Some of our abstractions are used to model physical reality, while others are completely adrift from that reality. For most of our existence, abstraction as a model to make sense of the Universe we found ourselves in was a key element of our survival. However, we are capable of taking abstraction to completely different realms. As Harold Abelson notes in the recording of his lecture on Structure and Interpretation of Computer Programs, software abstractions differ from physical systems in that we can know as much as we like about them. Software abstractions are idealized systems. The only constraint is the human imagination. In this regard, software abstractions share many of the attributes of mathematical abstractions.

But there is a danger here. We usually want our abstractions to tie back into our intuitions to some degree while they must also be able to operate at an acceptable cost. With no real constraints besides our imagination, we need to apply techniques to structure our abstractions in similar ways to what mathematicians do, but at the same time, we must remain firmly anchored in engineering. As such, software development inherits from mathematical theory building and concrete artifacts building. This is what I call Abstraction Crafting.

I will start with a number of portraits. Although the list of the people I selected might look arbitrary sometimes, it'll become clear throughout the document why I chose them.

António de Andrada

António de Andrada was a Portuguese Jesuit missionary monk born in 1580. In 1624, he left his mission in Agra, India, to join a group of pilgrims headed to Tibet. In his extensive accounts, where he documents his journey, he explains that at every snowy and frozen pass in the Himalayas, he convinced himself that it was the last one. Soon enough his guide left him, as the adventure was becoming too strenuous. Sometimes, de Andrada had to crawl on the snow to make progress. At times he suffered from snow blindness. But he was relentless. At one point he saw blood on one of his hands. He realized one of his fingers had frozen, causing a hemorrhage. After crossing more gorges than he had ever imagined, he eventually crossed the Mana Pass to enter Tsaparang, the capital city of the Tibetan western kingdom of Guge, becoming the first known European to set foot in that region of the world.

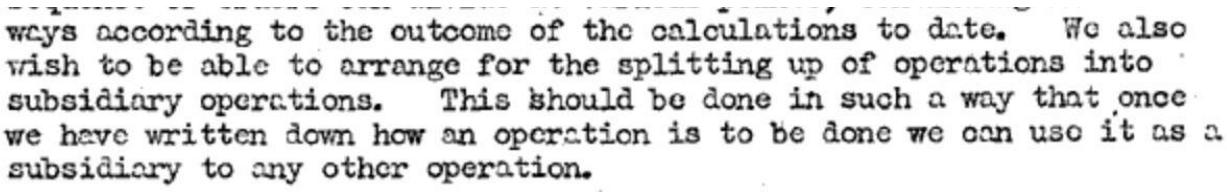


There are several lessons to take from de Andrada's journey four centuries ago. First, the human mind has a tendency to underestimate the amount of effort to achieve a goal. At every milestone, more difficult than the previous one, we hope it is the last. We want to believe it. But that ignorance is also bliss in the sense that we would probably stop if we knew what lies ahead. Last, no matter the hardships, keeping moving, being relentless, is what brings joy and reward.

In which ways does this story apply to the world of software? In more than you would think. In comparison to exploration, software is a recent adventure in the history of mankind, it is not even a century old. However, as software continues to eat up the world, it is becoming literally synonymous to progress. But surprisingly, the pace at which breakthroughs are made is chaotic, and very little literature exists on trying to understand the mechanisms at play. While de Andrada was facing physical pain, the equivalent in the world of software is intellectual hardship. That hardship and the anxiety it can cause weakens our will and we are tempted to become de Andrada's guide and surrender: there is nothing ahead, let's just be content with what we have. We just look around and settle. Worse, what lies ahead must certainly be unattractive. Sour grapes ensue. Resolve has vanished.

Alan Turing

It is 1945. Alan Turing is busy launching what would become mankind's biggest initiative since the invention of fire: software. That year he writes a report called "Proposed Electronic Calculator." He would submit it to the United Kingdom's Executive Committee of the National Physical Laboratory in February 1946 under the description "Report by Dr. A. M. Turing on Proposals for the Development of an Automatic Computing Engine (ACE)." In this important seminal document, Alan Turing introduces the concept of "subsidiary operation", later known as a **subroutine**:


ways according to the outcome of the calculations to date. We also wish to be able to arrange for the splitting up of operations into subsidiary operations. This should be done in such a way that once we have written down how an operation is to be done we can use it as a subsidiary to any other operation.

A "subsidiary operation" is a list of instructions which can be reused instead of being repeated over and over again. Using this concept, it is possible to give structure to a piece of software by removing repeated chunks.

The idea of repetition is central to Turing's contribution. Subroutines are a way to avoid inserting the same code again and again. They introduce order, they fight entropy. In the same report, he defines the role of software as automating repetitive tasks. Interestingly, before we used the term "algorithm" he used the expression "mechanical tasks", grounding software into the practical rather than the theoretical. Quite a statement from a mathematician.

In his idea of subsidiary operation, Turing introduces a separation between two pieces of code, one higher-level and one underlying. The first relies upon the subsidiary one. A critical property of the underlying one is that it is reused across higher-level ones.

We can identify a similarity with the process by which mathematics is progressing. After all, Turing's background is math. Math theorems are statements which have been proven true. They routinely rely on lemmas or other theorems. Nothing is ever circular, otherwise the very idea of truth could be questioned.

Of course the idea of subsidiary operation, or as we would call it today a subroutine, is still primitive, it does not yet assume any interface between the higher-level and the lower-level code. In that sense, it is also very similar to math. There are conditions to be respected for a theorem to be used, that's all.

But once a theorem has been proven correct, it can freely play the underlying infrastructure for multiple new theorems. Progress can be defined by pushing further what we hold true. Existing theorems are the shoulders of the giants we can sit upon to see further.

The pace of progress, the discovery of new theorems based on prior ones, is unpredictable. Sometimes it takes very little time, sometimes we have to wait for centuries.

Turing does not mention the ability to use subroutines within other subroutines, but we can safely assume that, as a mathematician, this was too obvious for him to mention.

A famous modern example of the usefulness of sitting on giants' shoulders is the demonstration by Andrew Wiles of Fermat's last theorem. It took 357 years for a mathematician to demonstrate a generalized version of what Fermat had written in the margin of his Diophantus Arithmetica book. We know that Fermat presented it as fact, while he most likely didn't actually have a demonstration. Andrew Wiles dedicated his entire life as a mathematician to a generalization of this conjecture, and he had to use an impressive amount of tooling to achieve his goal. By tooling, of course, I mean an extensive amount of theorems which are part of the layers of progress accumulated in the 357 years that preceded his 1994 breakthrough.

Let’s just conclude that Turing was certainly inspired by the reusable nature of math theorems to come up with the idea of subroutine. Turing says nothing about the process by which we can come up with subroutines while writing code. He does not say whether we just identify exact repetitions and factorize them like we do for algebraic formulas. He does not even say if we should rearrange our code to make these repetitions appear to make factorization possible. He is not talking about chunks of code, the hint he gives is in the word “operation.” It is a word which evokes a conceptual role for subroutines.

Math has no standard recipe for identifying concepts. In that sense it is an art. In addition, math involves an inherent creativity, a beauty found in its elegant proofs and structures, and a connection to artistic expression through patterns, proportion, and geometry.

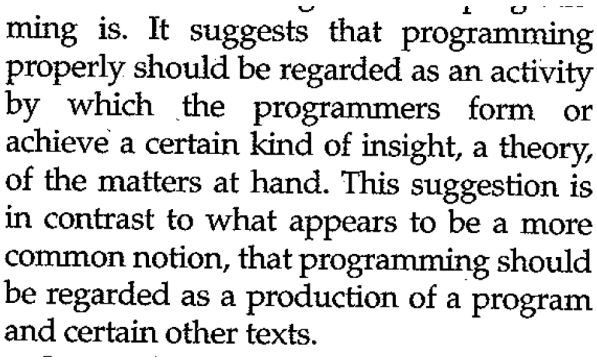
So, coming from math, it was enough for Turing to state that software is meant to automate repetitive tasks. We can only speculate he would have said about software itself what he said about math: “Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.” The word “ingenuity” carries the ideas of being clever, inventive and original.

With subroutines, Turing planted a seed, and left reflecting upon repetitions in software to followers. And that’s exactly what Peter Naur proposed in 1985.

Peter Naur

It is now 1985, a whole forty years after Turing's report, Peter Naur publishes an article called "Programming as Theory Building." While this article should be on every software practitioner's desk and taught early in the computer science curriculum, it is unfortunately only known by very few. In the title, the word "as" needs to be understood as an equivalency. It does not mean that programming is "akin" to theory building, not even that it occasionally involves theory building. It means that programming *is* theory building. All the time.

In this document attempting to "meta"-understand the elaboration of software, Naur explained how it should be approached:


ming is. It suggests that programming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand. This suggestion is in contrast to what appears to be a more common notion, that programming should be regarded as a production of a program and certain other texts.

We have to set these statements in context. Between Turing's words in 1945 and Naur's in 1985, a whole lot of progress had been made around software. When I was writing software in 1985, I had long left behind the subroutines I was creating in my assembler days. Namely, I was using a processor which contained microcode wrapped into machine language, this machine language was produced from a higher-level assembler produced automatically by the C high-level programming language using structured programming, types and functions. I was about to use Ada 83, which offered generic programming and very soon after C++ which added object-oriented programming to all of the above. In 1985 we had accumulated an impressive amount of progress.

Yet, in 1985, there was no general attempt at understanding the mental process around producing software. Worse, the general take on that, which still probably holds true to this date, was that software was merely about producing code. This is why Peter Naur felt compelled to write his piece.

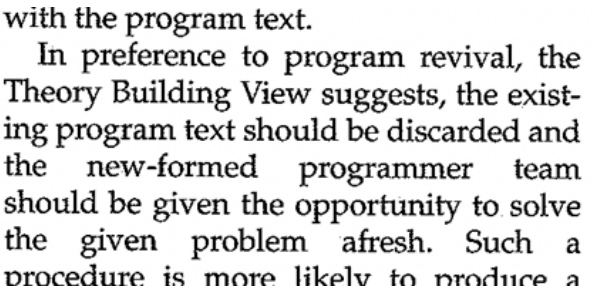
In this document, in short, what Peter Naur tells us is that to program is not merely to produce a program made of code. He uses a word that few would associate with it: theory. So what is a theory in the general acceptance of the word?

> A theory is a well-substantiated explanation of some aspect of the natural world, based on a body of facts that have been repeatedly confirmed through observation and experimentation. It's more than just a guess or speculation; a theory is a robust framework that integrates facts, laws, inferences, and tested hypotheses to provide a comprehensive understanding of a phenomenon.

In other words, a theory is what results from taking a specific view on a domain of the world, identifying repetitions, and finding a repeatable explanation and understanding for it. This definition makes the interesting use of a term familiar to Turing: repetition.

Naur explains that looking at software as applied science is wrong. Software is not the application of a predetermined theory. Its very nature is to *define* theories. This is very powerful and true.

Naur's enlightening Theory Building View is a purely intellectual human endeavor. It remains disconnected from the artifacts produced by programmers. Naur explains that the theory he is talking about stays as a mental construct of the programmers, carried by the programmers. According to him, it cannot be captured by documentation. He goes as far as recommending to trash the entire code when the team in charge disappeared some time ago and the work needs to be revived:



with the program text.

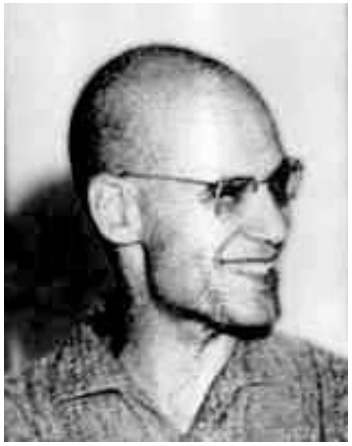
In preference to program revival, the Theory Building View suggests, the existing program text should be discarded and the new-formed programmer team should be given the opportunity to solve the given problem afresh. Such a procedure is more likely to produce a

He also explains that in general, there is no such thing as a scientific process to elaborate a theory. This seems to place a final word on trying to guide programmers. But, is he right? There was at least one person who had a system.

Alexander Grothendieck

The word “theory” used by Naur is both unexpected, striking, but also very general. Naur says that there is no general scientific process. Epistemology does not provide any help. But when we develop software, we operate in a much narrower domain. Even in math, one key person disagreed with the fact that there was no scientific process. Alexander Grothendieck, probably the greatest mathematician of the twentieth century, kept a correspondence with Jean-Pierre Serre, a founding member of the Bourbaki collective. The latter wrote the following on Grothendieck’s original approach to “making math”:

> Somewhere, you describe your approach to mathematics, in which one does not attack a problem head-on, but one envelopes and **dissolves it in a rising sea** of general theories. Very good: this is your way of working, and what you have done proves that it does indeed work, for topological vector spaces or algebraic geometry, at least...It is not so clear for number theory…



Somehow Grothendieck considers that the “lemma approach” in math, starting from the preparatory foundation, is the slow-grinding process that should be used. A lemma is a subroutine. Following this philosophy, Grothendieck promotes at a gentle “non-sporty” but persistent and tenacious bottom-up process whereby you set your eyes on a high-level target and you build tools—understand demonstrated results—(the most elementary being a lemma, familiar to mathematicians, and the most salient being theorems) that use each other, at a gradually higher level until the problem has been tackled (dissolved) by being a trivial result of the highest-level tools. We could say that math results slowly but surely stack upon each other to reach the targeted level. That approach has been qualified as “virile”, as it compares in the intellectual endeavor with constant and inordinate muscular force to rise the whole sea. It does not seem unreasonable to make a parallel with the software development process, and Jean-Pierre Serre's objection on number theory does not seem to apply to software which has more to do with Grothendieck’s body of work around mathematical structures. Grothendieck

introduced the powerful idea of topoi, used as “bridges” for connecting theories, in a sense not dissimilar to Naur’s use of the word.

Talking of topoi, and more generally category theory, there are additional links with abstraction. These links have been described by Robert Goldblatt in “Topoi: The Categorial Analysis of Logic”, Elsevier North-Holland Publishing Co., New York, 1983. Goldblatt writes:

> **The pathology of abstraction**
>
> The process we have just been through in identifying the notion of a category is one of the basic modi operandi of pure mathematics. It is called *abstraction*. It begins with the recognition, through experience and examination of a number of specific situations, that certain phenomena occur repeatedly, that there are a number of common features, that there are formal analogies in the behaviour of different entities. Then comes the actual *process* of abstraction, wherein these common features are singled out and presented in isolation; an axiomatic description of an “abstract” concept. This is precisely how we obtained our general definition of a category from an inspection of a list of particular categories. It is the same process by which all of the abstract structures that mathematics investigates (group, vector space, topological space etc) were arrived at.
>
> Having obtained our abstraction concept we then develop its general theory, and seek further instances of it. These instances are called *examples* of the concept or *models* of the axioms that define the concept. Any statement that belongs to the general theory of the concept (i.e. is derivable from the axioms) will hold true in all models. The search for new models is a process of specialisation, the reverse of abstraction. Progress in understanding comes as much from the recognition that a particular new structure is an instance of a more general phenomenon, as from the recognition that several different structures have a common core. Our knowledge of mathematical reality advances through the interplay of these two processes, through movement from the particular to the general and back again.

All this resonates powerfully with our domain. We even use the same vocabulary, although for slightly different purposes.

Bjarne Stroustrup

Bjarne Stroustrup, the inventor of the C++ programming language says that C++ is meant to "express ideas directly." Like Turing's "subsidiary operation" or Naur's "theory building", it means many profound things in a few words. In particular, it means that ideas can be conveyed by the code artifacts.

The root of this can be traced back at least to the idea of abstract data type (ADT), formally introduced and developed in the early 1970s, by Barbara Liskov and Stephen N. Zilles.

Of course, Bjarne Stroustrup means more than just ADT. Expressing ideas directly in code means using abstractions. Every software programmer is familiar with the word. It has taken a bad rap, and I'll come back to that. But let's use it nevertheless.

When Turing says that a subroutine is a lower-level reusable building block, he is actually talking of one of the first instances of abstraction. From the point of view of the artifacts using the subroutine, the subroutine is an underlying engine, capable of performing a dedicated task. There is a caller and a callee, separated by what we would call today an abstraction layer.

ADT pushed the concept further, but it was the 1970s. Since then, we have had object-oriented and generic programming, both powerful ways to make abstraction explicit and concise in code artifacts.

While abstractions are not sufficient to capture a theory in Naur's sense, it has been underlined by Donald Knuth in 1984, with his Literate Programming paradigm that there are ways to make programs readable by programmers who did not write them. It uses natural language embedded with source code.

While Literate Programming has not been embraced by a large community, it is clear that the combination of higher-level abstractions such as the ones proposed by OOP and generic programming, in combination with robust natural language commenting, we can today have

both the ability to express ideas directly, and transmit an entire theory through code artifacts. We do not need to rely on a continuous presence of programmers who transmit knowledge informally.

Expressing ideas directly can easily be misunderstood as a top-down process whereby you start with requirements as code and you gradually land the ideas on preexisting and low-level abstractions. That's what we are taught in computer science classes. The word factoring itself, also found in "refactoring," entails a top-down approach. In contrast, the Grothendieck approach, the bottom-up sea rising, is the non intuitive and correct way to build and formulate a theory through abstractions. At the end of the process, the layered abstractions look like what Bjarne Stroustrup calls "Turtles all the way down." If the top-level abstractions are general and high enough, no top-down process is needed. It is just a matter of expressing the ideas, the requirements, in the language of those abstractions. As we shall see, layered abstractions can be anything but slow turtles.

Abstraction

I haven't been able to find the word abstraction in Turing's legacy. I have probably not looked hard enough. My excuse is that very little of his works are available online in searchable format. The word abstraction is used only once in Naur's paper.

If we let go of the sciency word "theory" from Naur, and introduce the intentionally-vague word abstraction, while combining his opinion with Bjarne Stroustrup's, we can reuse and reframe his point of view:

> Programming within a specific domain is to conceive layered reusable abstractions which allow to give sense, structure, conciseness and robustness to the code. This allows us to express and understand ideas directly.

To some extent, the word reusable is not needed, as the very idea of abstraction should convey the idea of reusability. But it is easy to forget it, so it is a good reminder.

The word abstraction is a good one, as it ties with the idea of practice that Naur wanted to retain. He talks of programmers, not of analysts. In 1975, Fred Brooks wrote in "The Mythical Man-Month": "Thinkers are rare; doers are rarer; and thinker-doers are rarest." This was echoed by Steve Jobs in 1995: "The doers are the major thinkers." I have explained the orthogonal difference between manufacturing and mentofacturing in "Mentofacturing: Silicon Valley's Secret Sauce." Which word better than abstraction would convey the idea that it's practice which causes most of the thinking?

The word "abstraction" comes from the Latin verb *abstrahere*: to "pull" or "draw" away ("ab-" prefix). It contains the idea that it works from the ground up, by elevation. In contrast, the word theory would most of the time inspire the idea that it is top-down and merely ideas, while software abstractions are artifacts.

To the non-programmer who has used a spreadsheet, the idea of abstraction is easy to understand. A spreadsheet cell with a formula depending on other cells is an abstraction. In this example, understanding why abstractions stack on each other like turtles is easy: a formula can use other cells containing other formulas. The cell "sits" on top of the other cells it depends on. It is not obvious that there is a hierarchy, as a spreadsheet is presented as a table, but that hierarchy exists. No circularity is allowed.

A complete lack of abstraction is a spreadsheet with a single cell containing the whole formula. The advantages of abstraction in a spreadsheet, in terms of reuse, maintenance and identifying "bugs" are completely identical to the ones enjoyed by software programmers. Structuring a spreadsheet correctly requires a little bit more intellectual work, but it pays off.

Similarly, the idea of reuse, which is consubstantial with abstraction, is easy to grasp. If a cell already contains a formula that you need, there is no point creating another one doing the same task. You just use it. When writing code, for instance when a sorting algorithm has been programmed, a software developer does not need to develop it again and again every time they need one. The qsort function appeared in 1972 in Version 2 Unix and while it evolved, it has been used from a library ever since in the C programming community. The only point of writing more code is to write code that brings something new, never seen before. If a piece of code already exists to do a task, software developers just capture it in an abstraction and reuse it. Only ingenuity is left. Except in some rare circumstances where a software developer is looking for an API or some rare artifact, copying, pasting and adjusting code found on Stack Overflow by hand or through some AI-assisted coding tool is sloppy work which gradually degrades the quality of the work. That's the equivalent of inserting again and again similar bits and pieces of spreadsheets, hoping that it will remain maintainable. It won't.

I am going to take another example which, this time, is completely outside the world of technology: law. We have essentially two systems in the world: civil law systems are prevalent in most of the world, while common law systems are found in countries like the United States (except Louisiana), England, and other former British colonies. Civil law is sometimes called "Roman law" or "Napoleonic law". Common law uses an inductive, bottom-up approach where legal principles are inferred from individual cases. In contrast, civil law is based on comprehensive, written legal codes. Civil law is an abstraction of common law.

There is no formal definition of the word abstraction in the field of software outside of Abstract Data Types. The word "abstraction" is barely more precise than Naur's "theory." Its merit is that it is tied to code artifacts. Here are a few examples where the word can be used:

- Machine language is an abstraction on top of microcode.
- Assembler is an abstraction on top of machine language.
- Compilers which produce intermediary assembler code are an abstraction above assembler.
- Structured programming is an abstraction over unstructured code, in the sense that it incorporates it and builds over it.
- Object-oriented programming is often an abstraction over structured programming for the same reason.
- OpenGL is an abstraction on top of 3D graphic cards.
- ODBC is an abstraction over database management systems APIs.

Would this evolution of Naur's view integrating the word abstraction be more palatable to the modern programmer? The word theory sounds very sciency, and it probably contributed to the lack of popularity of his paper. Sadly, this rework would probably not improve it much. The word abstraction has taken a bad rap. Most professionals avoid it carefully.

Google Trends since 2004 show a troubled history on the interest around the word abstraction (category Computers & Electronics):

It is generating barely more than a quarter of the interest it had in 2004. And recently interest has curiously collapsed at the same time the fascination for AI and AI-assisted coding has grown.

Yet, the word abstraction is a good one. Examples of it are abundant, we have layered abstractions in 3D rendering, in gaming, everywhere as soon as we use libraries of algorithms and datastructures, literally every time we use any library. In the C++ programming language, Alexander Stepanov's Standard Template Library with three layers: algorithms, iterators and containers has been a brilliant contribution.

But in spite of the beautiful demonstration given by the STL and previous "theories," is there a consensus on abstraction in the software field?

Dissonant voices

Among programmers, abstraction does not always have a positive connotation. This situation has substantially deteriorated since the eCommerce talent shock around the year 2000. The sudden need of vast amounts of programmers has popularized the promise that software progress would materialize by making thinking less needed.

This being said, outside of the dissonance created by demand for programmers far exceeding supply, among famous programmers, notable ones have publicly expressed strong reservations against abstractions. Why would these, who include rock stars of programming, be that skeptical? We can distribute them in two categories, the talented re-coders and the distrustful software developers.

The people who belong to the first category usually re-implement abstractions which are part of familiar pieces of software and which have been collectively accepted for a very long while. Nothing is added, except excellent know-how to simplify and improve the execution. There is no surprise that this category is skeptical of abstractions, they are conventional.

The second category possesses creativity and contributes to crafting substantial, sometimes very famous abstractions. Its members have spent the necessary time to make the abstractions fast and frugal. Their technical attention is often downwards in the underlying layers, and not upwards. But above all, their excellence at the craft causes them to be very cautious about other people's abstractions. They will certainly not obsess as much on implementation details as they did. The second category is not against abstraction, it actually thrives on abstraction. It is just distrustful of abstractions not made by them. This is a variant of the famous "not invented here" syndrome.

The second category brings an important point, using other people's abstractions requires that you trust that the person in charge of the underlying abstraction. And to be fair, there are many more possibilities to come up with bad abstractions than good ones.

Of the two categories the first one is the most dangerous one in an organization, as their members will often go out of their way to prevent abstraction in their organization. The situation is even worse when they are in charge.

This brings us to another reason for distrust, which is not performance-related, but to the ability for abstraction to do the exact same work as it would be done manually.

Abstraction is not automation

In the heroic times, when we did not have higher-level programming languages yet, some solutions existed to produce lower-level machine code from an input barely better than assembler. It was called “autocode.” The name itself suggests that at the time, the idea of abstraction was not completely clear, and mere automation was the goal.

When people moved from assembly programming to higher-level programming languages, some were in pain. Not just because they had to embrace new paradigms, but because they were mentally doing the work of the abstractions proposed by higher-level languages while using them. This was painful. Learning to let go is hard. The only motivation can be found in looking up, not down.

Of course, like automation, abstraction incorporates the idea of each abstraction tool being able to execute tasks for you. If the abstraction is not doing the work, you are. For an assembly programmer, using a C language function call limits freedom, and can trigger the suspicion that it won’t replicate the same job they would have done by hand. This suspicion is justified, but it has obviously not been a reason to reject C function calls. This debate has been over for more than half a century, settled in favor of function calls abstractions.

This is a very common misconception about abstraction. Abstraction is not automation. Moving from a situation where you produce code artifacts by hand to a situation where you elevate abstraction does not mean that the exact same code will be executed. As Peter Naur put it, abstraction is theory building, and multiple theories can be produced for the same outcome, some better than others.

The fact that a new abstraction does not automate preexisting practices down to code is a frequent objection about abstraction. This is completely missing the point. It is just an excuse for the unwillingness to evolve and understand an unfamiliar theory.

The automation around code insertion, like proposed by AI-assisted coding tools, ignores the necessity to find a theory, and it automates the increase of entropy, more mundanely “slop,” pulling in the opposite direction of abstraction and its benefits. But we’ll address maintainability later. Suffice to say that abstraction is precisely what the current state of AI is unable to do.

Abstraction is higher-order compared to automation.

Abstraction crafting is not refactoring

A very common misconception among practitioners is that abstraction emerges from code and is the result of essentially mechanical refactoring. Abstraction can indeed be as mundane as a simple and general function. However, splitting a very long function in two separate functions to make it shorter, and distributing its contents in two subsidiary functions which have half the code size is absolutely not abstraction crafting. You have to ask yourself whether it is making the code closer to Stroustrup's direct expression of ideas.

If we read Turing carefully, he is actually not talking of a process whereby we would notice in a set of code artifacts some identical pieces of code, and we would mechanically factorize them and set them aside for reuse. He is talking of subsidiary "operations". An operation is a conceptual task, not a mere list of instructions. Abstracting is not meant to save on code size.

Turing is elevating the understanding of what a piece of code performs to promote it as an "operation". Turing is a mathematician, he is very much at ease with Naur's idea of a theory. He is very familiar with the need to assign proper names to concepts.

Good programmers are all very meticulous. And meticulous they can be to a fault. They would never copy-paste the same code over and over again, because it "does the trick". When they encounter repetitive code, they are triggered. This talent is their enemy in the case of abstraction. The temptation is huge to factorize. A popular motto is "don't repeat yourself". Doing so, without thinking much, latent abstractions are obfuscated, hidden from view. Factorization is the enemy of conceptual elevation. Excellent programmers learn to control their meticulousness (and not let their meticulousness control them), and accept to wait patiently for higher-order abstractions to emerge.

The best proof that abstraction is not ad-hoc factorization lies in Naur's article. It requires building a theory. It entails intense thinking, intuition, inspiration and ingenuity. Factorization can be done by a handheld calculator. It is purely mechanical and requires no talent.

Factorization can lead to creating functions, and functions are technically abstractions. So it means strictly speaking that there are actually good and bad abstractions. Some famous people even profess that there are only bad ones.

Note that abstraction delivers everything that factorization does, including reducing code size and as a result decreasing the amount of testing and maintaining of the code.

Abstraction does not rise from increasing conciseness

Conciseness is a clear benefit of good abstraction. The code resembles a direct expression of ideas. The need for commenting is greatly reduced as the code reflects the intent.

A very common rookie mistake is to believe the reverse too. While abstraction brings conciseness, conciseness does not give birth to abstraction.

Working hard at reducing the size of a piece of code has nothing to do with seeing abstraction emerge. It can even be speculated that it has the exact opposite effect, making code irregular through ad-hoc tricks which hide the potential for identifying latent abstraction.

Even the conciseness benefits of abstraction are not a goal in themselves, they are just a byproduct. Abstraction offers productivity, but software development productivity is not counted in the number of characters typed in source code. As everybody knows, the same goes for performance, shorter code does not always run faster.

This is powerfully explained in this quote from Edsger W. Dijkstra, taken from “The Humble Programmer,” his 1972 Turing Award lecture:

> The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.
>
> In the case of a well-known conversational programming language I have been told from various sides that as soon as a programming community is equipped with a terminal for it, a specific phenomenon occurs that even has a well-established name: it is called “the one-liners.”
>
> It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question “Can you code this in less symbols?” - as if this were of any conceptual relevance! - or he just asks “Guess what it does!”
>
> From this observation we must conclude that this language as a tool is an open invitation for clever tricks; and while exactly this may be the explanation for some of its appeal, viz. to those who like to show how clever they are, I am sorry, but I must regard this as one of the most damning things that can be said about a programming language.

There is no place to hide

Naur's paper is powerful. It states that programming is theory building. It does not mean that some parts of programming are theory building, while others are mechanical mile-long insertions of script-like code.

Following our evolution of Naur's recommendation, and the idea that abstractions embody theories, it means that there is no such thing as boilerplate code in proper programming. If you deviate from theory-building-abstraction-everywhere, you must constantly be on alert to identify a nascent abstraction.

If you fail to follow this process, as Naur points it out, you lose safety and maintainability, because entropy will increase. And the theory of abstraction, from the 1970s, says that you'll lose productivity by losing conciseness too.

Abstractions are everywhere, below and above abstractions. But don't be afraid, you are doing abstractions all the time. When you write a type or a function, you are actually writing an abstraction. There are small and big theories. So you should pay attention.

There seems to be a domain which is insulated from abstraction, it has a name: scripting. A typical example of scripting is most of what DevOps assets do. Similarly, people in charge of build write a lot of scripts. Not all scripts are devoid of abstraction, you will find scripts with functions inside, even types. When these are missing, a big question is whether these activities require abstraction-less scripts or whether the people writing them have never been taught the value of abstraction. Assign an engineer familiar with abstraction to DevOps and build, you'll see an entire population of abstraction arise. So, there is no reason to believe that DevOps and build cannot benefit from the benefits of abstraction, with only one caveat which is potentially the quick turnaround for code written in these domains. But be careful, the frequent need for updates is also the result of the lack of abstraction, so the reasoning is circular. DevOps and build are insufficient examples to say that there are domains where abstraction is useless.

If you can't name it, it is probably a bad abstraction

This is a consequence of Naur's paper. In Science, and especially in an activity neighboring Computer Science, math, it is well-known that the ability to name things properly is essential. Names don't come last, they come first.

Alexander Grothendieck based much of his career on the idea that we must name things in order to begin to glimpse them. This principle is even rooted in Judeo-Christian culture; the Bible itself, in Genesis, describes how God begins by naming the elements of creation. On the first day, he names the light and the darkness. On another day, he names the sky, the earth, and

the waters. He gives names to the stars. Later, Adam is tasked with naming the animals. This act of naming is the first step toward being able to envision something. A powerful example is also given by the numeral “4”. Ancient societies used to have “1”, “2”, “3” and “many”. Numbers more than “3” had no specific name. Every word we use is an abstraction. It’s meant to encompass a whole set of usages, and when you pronounce it, you do not need a long explanation, people sharing the same vocabulary understand immediately when the word entails.

If you attempt to create abstractions through factorization, you will probably not be able to give a name to the factor you will create. The same goes if you break a long function into two smaller ones. It will not be possible to associate the result with an articulable concept, which is part of a theory. Just from the angle of naming, factorization is most of the time a bad process.

Similarly, if you come up with one or several abstractions and you cannot figure out a proper name for each of them, it means you do not have a mental structure of how they are to be used or how they interact. You do not have a theory, you do not have abstractions in hand. Naur would probably say that you have nothing robust.

So, trying to name abstractions is the very first step to undertake. It is the first acid test. Get help, some people are better at naming than others. If you can’t explain it to others, it is a bad sign too.

There are plenty of examples of good names: multitenancy, iterator, container, referential integrity. Having a good name does not guarantee that the abstraction is good. There are examples of bad ones too, take “document database”. The word “database” is well chosen. However, “document” is very misleading, it gives a feeling that the database stores office documents. Worse, it hints at the database storing a representation of things, rather than the essence of things. Nowadays what used to be called a “document database” is called a “NoSQL database” instead. That’s not better. “NoSQL” means “Not only SQL”. At the same time the same databases have to explain that they offer a query language that they invite you to work with. The jury is still out for a better name.

Don’t be a Flat Earther

Abstractions work in layers, if you reach a point where you think the theory is complete and there is nothing at a higher elevation, you are probably wrong. Until you are able to express ideas directly, there is still space to fill.

Going up is hard, it requires thinking. Don’t be a Flat Earther, there is always something beyond. A sign that you are a Flat Earther of abstraction is that you can only conceive helpers. A new abstraction sits on top of others. It is not a set of helpers at the same level around what you already have. Elevating abstraction requires a shift.

Here is an example, the Linux filesystem. The filesystem with its inodes is an abstraction. It sits on top of the disk driver which is the underlying abstraction. Imagine you do not yet possess the idea of a filesystem, and you operate only at the level of the disk driver, writing blocks to the disk. Imagine you want to remove a file, you are probably going to zero the bytes of that file somewhere. Elevating the abstraction to a Linux filesystem, removing a file leaves the file on the disk. The bytes stay there. They sink. The reference for truth in terms of file data is not the disk any longer. It's in the inodes. The filesystem has elevated the interaction. Adding helpers to zero the bytes on the disk through the disk driver is not an abstraction.

Accept to let go, do not recite a theory mechanically as if it were the ultimate one, look up, take another angle, stay alert. Realize that you are already sitting on top of a myriad of abstractions, thinking that you are at the top of the hill and there is nothing beyond is futile.

Beware of amalgamation

The oldest among us will remember the Open Systems Interconnection (OSI), a reference model proposed during the late 1970s and breaking down communication systems in seven abstraction layers. Seven! This was the culmination of a lot of know-how, it literally required acumen. It would have been easy to amalgamate a few of the close layers. But no, making a distinction helps building systems. And note the usage of the expression "abstraction layers".

Let's go back to the C++ Standard Template Library, invented by Alexander Stepanov. It was a breakthrough not because it elevated abstraction above what we knew. It was a breakthrough because Stepanov's sharp mind caused him to insert an interstitial abstraction layer between our familiar algorithms and data structures. Which software practitioner hasn't been through Computer Science 101's lecture on data structures and algorithms? Before Stepanov, we were building our algorithms on top of data structures, using their specialized API. Stepanov's contribution looks superficially simple, and unimportant, making algorithms rely on iterators, and classifying iterators in families. This eye-opening suggestion suddenly allowed us to write algorithms to a large extent independently from the underlying data structures, provided they offered iterators of the right family. What he showed us is that we had always amalgamated two distinct abstraction layers. Adding subtlety and a third layer between our two familiar ones brought a lot of benefits.

Another example of interstitial abstraction where a new abstraction has been added between two familiar ones is virtualization. Whether you are using a virtual machine interpreting code or some other implementation based on containers, adding virtualization means adding a layer between your application and the operating system it runs upon. Virtualization can even add two interstitial layers, with a second operating system running on top of the underlying one.

How about taking a controversial modern example? The world has been abuzz for a long time with the potential of the Internet of Things (IoT). Devices are the new users. There are billions of them. They open up new fields of opportunities. They can prevent downtime, avoid

catastrophes. Working groups are busy standardizing network protocols. This is important because most devices are not capable of high-level OSI features. But this diverts the attention on the final goal, making applications capable of integrating devices as the new users. When your focus is on low network layers, the temptation is great to be a Flat Earther and implement applications within the low abstraction layers. By application I do not even mean OSI level 7, which is purely on information exchange. I am talking of the application receiving and sending the communication.

These examples teach us to take a loupe before settling on abstraction layers, a bit like examining with a high-resolution spectrometer the hydrogen spectrum. It exhibits fine structure when examined with a high-resolution spectrometer. The red H-alpha line appears suddenly as a closely spaced doublet. This splitting, though very small (approximately 0.016 nanometers), is evidence of the electron's spin and the spin-orbit interaction. Not a small result. The same goes with abstraction. With enough finesse, it can be a revolution.

They teach us to have a sharp mind and be very rigorous. Adding an interstitial abstraction is less powerful than adding a good abstraction at the top of a stack, but it can still open a world of possibilities.

Resist deconstruction

Now you have an abstraction. It is very tempting to fuse it with its higher-level or lower-level abstractions. The perceived benefits are multiple, you get additional freedom, you might save a few processor cycles. It seems you are optimizing. Additionally, deconstruction requires no ingenuity, it is completely mechanical and like taking a vacation of the mind, while satisfying some kind of obsessive mechanical tendency. The time spent by a software developer on mechanical tasks is always substantial, but when it is about dealing with abstraction, mechanical thoughtless activities are the enemy. The path of abstraction is a straining one, an uphill battle, deconstruction is the easy slippery slope in the opposite direction.

Removing an abstraction makes the code more complex. With many more options that are harder to explain, potential users of the abstraction won't be willing to spend the time understanding them. Abstractions are meant to be reused. They must therefore be understood. Deconstructing pushes towards the "747 cockpit" syndrome. So many options that nobody except a few will be capable of using the deconstructed outcome. Simple but not simplistic abstractions are an illustration of the "KISS" principle (Keep It Simple, Stupid). There is actually nothing stupid in that; simplicity is harder than complexity.

With deconstruction, your code will become more difficult to maintain and to test. Deconstructing manages code, not the theory. The drawbacks are exactly symmetrical to the benefits.

Don't cede to vertigo when facing stacked abstractions. Some of the best abstractions are simple layers, which are the most susceptible to tempt a deconstruction tendency. Simple does

not mean weak. After all, the entire body of math proofs we have, including the most complex, are just sequences of obvious statements. It's the accumulation of evidence which shows its prowess.

So, don't self-indulge in deconstruction, don't be lazy. Do it only if what you deal with is a bad abstraction and you are waiting for a new one to emerge. Obsess then about reconstruction.

Good abstraction is general

Naur's use of the word "theory" embraces the idea that it encompasses and explains an entire selected domain. The equivalent in software is that the abstractions you create must be effective in the sense that they must cover a vast field of applications.

This is what Naur expresses when he says that a theory must be "robust". He is not referring to bugs at all. It's not the robustness of the code that he has in mind. He is referring to the ability for abstractions to be general and applicable to an entire articulable and substantial domain. Every time a new use case for the abstraction arises, the theory behind it should remain stable, provided the use case is within the domain. To some extent, robustness is also the relative stability of the theory when the domain itself stretches a bit.

Of course the temptation is to extend the generality of abstractions. When you have a STEM background, it is almost a Pavlovian reaction. In the field of software, it is often counter productive. When object-oriented programming was in full swing, quite a few people invented the "Class class", meaning that suddenly you would have only one type, the "Class", and it would be associated with functions allowing you to add properties to it. This is an extreme example of collapse of abstraction when it is over generalized. It leads to lack of safety and decreased performance.

An example of widespread abstraction which is promoting a non general approach is the RESTful set of principles. RESTful promotes an interaction with data reminiscent of LDAP based on "resources". It is a simplistic view of data modeling. Very few applications serve resources to their client systems. Even the simplest back end applications contain business logic and relatively sophisticated data modeling. RESTful does not want to see that. It's not stimulating the mind, it shrinks it. It is a bad abstraction.

Another bad abstraction which is often commensal with RESTful is GraphQL. SQL is already a language which was born at the dawn of the 1970s when abstraction was not yet well understood. SQL does not allow for reuse and basic abstraction mechanisms like types or functions. Around 2012 Facebook, a company which is not building business applications released GraphQL in order to dynamically make selections within web services replies. The business application community quickly jumped onto the idea that all business logic should be retrograded to GraphQL, which is easier than writing abstractions the usual way. Everything becomes old COBOL-inspired SQL of the early 1970s.

Among good abstractions we count time series. Time series are used in financial applications where historical data and timestamps are to be archived, inspected, worked upon, processed. They are a powerful and good abstraction in the sense that they even apply outside of their initial domain. Take Google Borgmon, Google's home grown data center monitoring application. It uses time series to store a wide variety of events and trigger automatic action when specific configurable situations are encountered. Time series can also be applied to the Internet of Things, when properly done, or even Messaging systems.

The generality of an abstraction can even go beyond its author's intent, to the point of surprise. An example of this is C++ templates. When Todd L. Veldhuizen showed formally that C++ templates are Turing-complete, it unveiled a powerful property of C++-based generic programming that went beyond the intent of Bjarne Stroustrup, the inventor of the C++ programming language. Following Peter Naur's point that programming is theory building, a good theory is a theory which allows to describe much more than its initial intent. This is quite well-known in Science.

Good abstraction can be explained succinctly

As a first statement, if you can't explain it, it's not an abstraction. This sounds pretty radical, but not being able to explain an abstraction at all is a situation more frequent than it looks. It is often the result of mechanical refactoring. The result is ad-hoc, and does not correspond to a consistent set of principles.

The purpose of abstraction is to offer additional productivity by "juggling with one-ton boulders". In other words you should not perceive the weight of what you are handling, wherever you are in a stack of abstraction. The purpose of the abstraction you are manipulating should be clear, and the complexity should be sequestered within.

The start of this is to name every component of an abstraction properly, so that its intent already appears reasonably clear just by naming it. Of course, simplicity does not stop at naming, it also encompasses the services the abstraction renders.

If an abstraction cannot be explained succinctly, it might mean it is the aggregation of several abstractions which have not yet been identified. It can also mean that it needs to be pruned to produce something simpler and sleeker. A good example of fatty abstraction is one which is not orthogonal enough and offers multiple ways to achieve something with very little relative gain between these various ways. As a simple illustration, "operation A" and "operation B" are available within an abstraction, but "A then B exists too." Just offer A and B, let abstraction usage combine the two. Concision is not abstraction.

Of course, “succinctly” needs to be understood as a relative word, with respect to the complexity of how the abstraction is implemented. The more succinct the explanation in comparison to the complexity of its implementation, the better.

Good abstraction is simple to use

Explaining an abstraction in a few words so that people have an immediate mental model of what they do, even if it is approximate, is essential. But it is only the first step.

The second step is to make actual usage of the abstractions simple. If your abstraction looks like a Boeing 747 cockpit with an explosion of parameters, it means that you have failed. Abstraction is not supposed to replace complex code with complex abstraction usage.

In case you end up with an abstraction complex to use, you can consider a taxonomy of simplifications. This technique has merits but its drawbacks too. If the taxonomy is not general enough and you have to add more members every time you face a few new use cases, you have failed again. Beware of the combinatorial explosion of the parameters.

Good abstraction unleashes creativity

If your abstraction makes creating new abstractions on top of it very difficult, you have a bad abstraction. On the contrary, good abstraction unleashes creativity. It is liberating.

Take the example of relational databases. In the last 40 years many people have tried creating abstractions on top of it. There is too much boilerplate on top. None of it has been satisfactory, including the ones automating boilerplate, such as the so-called “Object-Relational Mappings” (ORMs). There are very good reasons to believe, beyond the blatant proof of 40 years of failure, that relational databases alongside SQL are very bad abstractions. There is one simple reason for that, they do not authorize reuse of business components on top. Every new application has to be bespoke and can share nothing with previous ones, even if they belong to the same business domain. The only possible integration is through microservices, which is very weak and disappointing. RDBMS are one of the very scarce abstractions plagued by this severe limitation. Abstraction is capped. This causes a lot of contempt from database vendors for their own customers, as the customers are condemned to form large teams, constantly reinventing the wheel. Computer science within databases, neverending repetitive boilerplate outside.

Although document databases, now known as NoSQL databases allow building abstractions enabling the creation of reusable abstractions on top of them, they have never been pitched as such. They were pitched for efficiency and to store things. They do not need to cling to a simplistic data model like relational databases, but they owe some of their success to the

understanding that no data model is actually needed. They are good abstractions but they were not adopted because of that.

Take game engines like Unreal. They have created an explosion of creativity. They even have applications beyond entertainment use cases. These are good abstractions.

Good abstraction offers clear performance control

A good abstraction is not completely a black box. Its intent is not to guarantee that any use of the abstraction will yield good results.

Asymptotic complexity like in the big O notation is just one aspect of it. Beyond computer science complexity based on characteristics of the data given to an abstraction, a good abstraction offers a methodology for the user of the abstraction to understand and get fine control over how the abstraction will behave.

Take my Metaspex product which allows defining ontologies which will persist in document/NoSQL databases automatically. The underlying abstraction offered by Couchbase or MongoDB performs best when the database is able to retrieve entire documents containing multiple pieces of data in a single I/O request. This is why Metaspex allows people defining ontologies to distinguish object to object relationships within the same document (own-a) from the ones between two distinct documents (link). When describing the ontology, the software developer can finely control the amount of I/O to acquire or save the data. When using indexes, Metaspex guarantees that data is never accessed unintentionally by making full collection scans.

In that sense, SQL is a bad abstraction, because you can only guess what the SQL execution will do with your SQL code. It is only after testing performance and analyzing complex execution plans that you'll know whether the "SQL abstraction"'s cost is acceptable or not. If it is not, you can give index hints, but there is no guarantee they will be used. Your control of performance is limited by "hints". You are sometimes offered tools which make index creation recommendations to run the SQL statements you give it. But you have no guarantee that if you follow these recommendations the SQL statements will work better. Worse, you have no guarantee that different SQL statements will end up using the exact same indexes, and you might end up maintaining very costly indexes unnecessarily. You have no way to globally reformulate your SQL statements to properly share indexes. It is dark magic to a great extent.

Similarly garbage collectors are bad abstractions between code and RAM management. Their cost is not well identified and usually underplayed. We even hear sometimes that a piece of software using a garbage collector is faster than one without it, and the amount of RAM consumed is identical or that RAM has no associated cost. Another cost is more subtle, the effect of the GC kicking in on the fluidity of the application behavior. On both points, anything goes to be evasive. And that's the point, a GC is evasive about costs. In that sense it is a bad

abstraction. It is interesting to note that many of the supporters of the stance “GC is free” quickly adopted the Rust programming language which uses no GC, and suddenly professed that GCs are expensive. A good abstraction publishes clearly its cost, and you know precisely what you pay for.

Good abstraction is frugal

When an abstraction is properly used, leveraging the right controls, it must perform as well as code written by hand without it. In other words, using abstraction should not create overhead. This is especially hard to do when abstraction is used, as it should be, in layers. The reader may be familiar with the expression “curse of abstraction”. Very few programming languages offer zero-overhead abstraction mechanisms. Among the ones offering a large set of abstraction mechanisms and being in current use we count essentially only C++ and Rust. Even using C++ or Rust does not guarantee that the code is written with enough care to be zero-overhead. It just means that it is possible.

So, when developers did not use a zero-overhead programming language or were faced with bad abstractions, they concluded that abstraction itself is affected by a curse that prevents it from working in layers. The more layers you add, the slower the overall program becomes.

It is actually an exponential law, if N is the number of layers, 1 is the performance reference with handwritten code, and C is the average additional cost of each layer, the cost of the stacked layers is $(1 + C)^N$. With the number of layers growing, the overall cost grows very fast.

The only perfect solution to that is that C is zero. In order to obtain that, the first step is to use a zero-overhead programming language and work meticulously on the code within the abstractions so that they truly match handwritten code from a performance standpoint. It is clear that this is higher-order programming in comparison to inserting miles of linear code.

The zero-overhead principle can be summarized as the saying “You don’t pay for what you don’t use.” While it might sound petty or penny-pinching, it is not. This small principle is instead at the service of a grand plan, at the base of the ability to layer abstractions on top of each other, as cumulated cost follows the exponential law above. The C++ programming language where the principle originated allows for instance to write “if” conditions which are evaluated at compile time, enhancing greatly the ability to express abstractions which undergo valuable compile-time optimizations eliminating the need to pay for what we do not use. Of course if the C++ keyword “constexpr” is missing, and the person crafting the abstraction is careless, the condition will be evaluated at run time, the price will be paid, and the general sloppiness of the code will lead to limited stacks of abstractions or deficient performance.

Of course, the lower, or in other words the more “foundational” the abstraction, the more care must be taken with performance. Picture performance as the size of the turtle. The smaller it is, the more difficult it is to stack up more on top. You surrender your ability to express ideas directly as it requires the highest level possible. And beware of the flat earther bias, you never know how many layers you could add on top of an inefficient abstraction, even if you think at the moment it is at the top of the stack.

The programming language is one of the foundational abstractions. If you do not use a programming language offering zero-overhead abstractions, or if you use one and are not careful, you implicitly accept to limit your ability in benefitting from abstraction. Here also, you surrender your ability to express ideas directly.

I can take an example from Metaspex, the framework proposing high-level abstractions for business applications, starting from the highest abstraction to the lowest:

1. Web services rely on algorithms.
2. Algorithms rely on iterators.
3. Iterators rely on Metaspex referential integrity.
4. Referential integrity relies on the ontology expressed with Metaspex abstractions.
5. The ontology relies on relationships.
6. Relationships encompass “links” describing object-to-object relationships across documents.
7. Links rely on a database driver to load the documents containing the target object.
8. The database driver relies on the database specific API.

We have 8 layers of abstraction in this case. We can describe a similar stack with “own-a” relationships for object to object relationships within the same document. These relationships rely on smart pointers doing reference counting.

If you want to replicate handwritten code performance and at the same time benefit from the engineering productivity, safety and maintainability that abstraction procures, you have little choice but using a zero-overhead programming language such as C++ used by Metaspex.

A frequent misconception about abstraction is “what can do the most can do the least”. An interaction between Philippe Kahn, the founder of Borland and a computer science researcher highlighted that powerfully. In the early 1990s, Philippe Kahn had just made a demo of Turbo C++ and programmed in real time an ellipse class deriving from a circle class, adding an independent axis. While this is questionable, a computer science researcher in the audience objected and said that he would do the exact opposite: a circle is an ellipse, it just happens to have the same little and big axis. The ellipse, according to him, should be the base class, with the circle class deriving from it, the exact opposite of what Kahn had shown. What that person was suggesting was a bad abstraction. An ellipse can do the most and also the least, being a circle. It is not frugal to describe a circle because every circle would be bloated and would have two axes. If you follow the principle, your abstractions will be bloated, and you won’t be able to stack them in layers to achieve the ultimate goal of the direct expression of ideas with realistic performance.

A modern example of all this is offered by two popular tools, Kubernetes and Kafka. Both are abstractions. Because there were mass movements towards the use of these tools, the business applications community has implemented them blindly everywhere, whether they were useful or not, whether they were implemented for their intended purpose or not. Kafka is intended for event-driven and streaming use cases, but it can also do queueing. Let’s use it everywhere we need something asynchronous! This is abstraction misuse and even abuse.

The talent shock around the year 2000 caused by the sudden surge in the need for software development talent due to eCommerce has cast a persistent shadow on performance. To this date many people still say that “machines are free,” justifying any sloppy work, conveniently forgetting that user engagement is first explained by short response time and that cloud companies are growing thanks to ever growing machines footprints. With a disinterest in performance, the talent shortage has caused the industry to neglect abstraction. One sad consequence is that progress in business applications has stalled for at least a quarter of a century.

Good abstraction is a method

We’ve heard of the word “methodology”. A methodology offers a way to approach problems, a guide to think. While the word methodology does not directly relate to abstraction, it precedes abstraction. An abstraction is an automated implementation of a methodology.

Take UML, a diagram-based methodology which was meant to help analyze and communicate on what a business application is doing. The goal of the founders was to turn it into a tool to produce the code artifacts automatically. It spectacularly failed at that. One of the reasons was

that it did not apply Grothendieck sea rising method, it was top down, starting from the business analyst and attempting to land. The landing failed.

An abstraction is both a method and its execution model. This is already in the “theory” word used by Peter Naur. A theory is a way to dissect a problem alongside intellectual tools. Articulating these as abstractions ties the theory to code. Abstractions make theory land.

Assuming we have a high enough stack of abstractions, the ideas do not precede their expression, they are guided by the tools themselves. The abstractions help the software developer ask precise questions from a business analyst for instance. In many instances the business analyst won’t have the answers, and business decisions will have to be made. Abstractions give structure to the interaction, highlighting the fact that writing software is not a one-way process, but instead is a two-way one.

This is why the modern approach of starting with “user stories” which do not invite participants into abstraction can be detrimental to the production of software. If user stories are considered an unquestionable start which will percolate down to code, the opportunity to make what is desirable meet what is technically possible is completely missed. Desires are implicitly limited by our mental view of what is possible, while what is technically possible, through upfront dialog, can lead to unexpressed unfulfilled needs which can be realized and become the unique selling points of a solution. Abstractions with their method allow the proper interactivity with users, so that the optimal meeting point is found. Software developers are active participants, even drivers of the process, rather than passive doers. Questions are more important than stories. All this was summarized provocatively by Steve Jobs who said that users must be given what they need not what they ask. He also said in 1995 that “the doers are the major thinkers”, doubling down on a 1975 statement from Frederick Brooks: “Thinkers are rare; doers are rarer; and thinker-doers are rarest.”

If an abstraction does not come with a method, it’s not a theory general enough. You have to work harder to devise higher-level abstractions. Stack up abstractions is a monotonous process which peaks unavoidably after a while by separating the what and the how, and the usage of high-level abstractions is undiscernible from the expression of the needs, the requirements. With just a little bit of exaggeration, as comments are still necessary, the requirement is the code is the documentation. The last bit about documentation is a direct consequence of Naur’s article, as he stresses the fact that a significant responsibility of theory is precisely the explanation of a problem at hand.

This is why abstraction is a two-edged sword. As it shapes the mind, once adopted, it is very difficult to let go to regain one’s mind flexibility and acquire a glimpse of higher-level abstraction. So it is both good to let one’s mind be formatted by abstraction, and it is important at the same time to make efforts to keep a curious mind and the ability to be iconoclastic (literally “break the icons”). It is extremely difficult, as the human mind likes to dwell and rest on systems.

The best example of the danger of this has been given over and over again by Grace Hopper who denounced the conventional behavior of the engineers working on business applications. She was known to say that “The most dangerous phrase in the language is, 'We've always done it this way.'” It is true that since the relational model, the normal forms and SQL became mainstream, engineers, including the ones with a computer science degree and an extensive experience of object-oriented programming tend to switch into “recitation mode,” just uttering word for word some principles they learnt by heart, irrespective of their interlocutor and like feverishly starting a rosary prayer, especially when confronted with refutation.

The theory part in a good abstraction is mesmerizing. The mind is fascinated by what could be called “beauty.” It causes it to neglect the other hard and consubstantial parts of a good abstraction. It is very tempting to become a missionary of that theory, and propagate the good word. The thing is that there can be several competing theories for the same domain. And sometimes the competition is just artificial, while a combination of several of these theories is the answer. A famous example of this is Alexander Stepanov’s harsh criticism of object-oriented programming and the promotion of his views on generic programming. He says that OOP is a “joke.” This is because he neglects a large part of the programming domain and focuses only on what is commonly called “processes”. A small venture into abstract syntax trees and grammar production rules is enough to convince oneself that OOP including polymorphism is essential. There is no way to deal with this problem with generic programming only. This generalizes to any data centric application handling complex data. An example of this is the survey industry. A questionnaire is a highly polymorphic object, it contains highly varying question types. The survey industry is still struggling to this day with the mapping of these complex objects onto relational databases which are not meant to deal easily with polymorphism. The concept of polymorphism did not even exist when the relational model was invented.

The mesmerizing nature of abstraction causes mass movements, amplified by missionaries, which in turn increase the mesmerizing effect in a feedback loop. We have all heard the sentence “It’s progressing” to disregard abstractions’ lack of frugality. The syllogism in that sentence is obvious, a progress in performance does not guarantee that remote goals will be attained. As a matter of fact plateaus are the most frequent outcome. It’s the appetite to believe that creates these baseless arguments. We must not lose sight that it’s the abstraction themselves which are breakthroughs. The way they are implemented usually does not enjoy the same positive disruptions. In a sense, abstraction is innovation, while clinging to them is convention, including when they are coming from mass movements. The convention on innovation is still 100% convention and 0% innovation.

Also don’t be confused, abstractions involve a method, every method is not an abstraction. Take the example of the so-called “Gang of Four” design patterns book. It is often mistaken with a cookbook, which it is not. It is a method, but it is not an abstraction. Even more, it is an eye-opening book increasing the malleability of abstraction crafters. Read it, but don’t obsess with it, and of course do not expect to derive reusable artifacts from most of it.

Abstraction can require a change in perspective

It is established that writing software is putting together theories captured as abstractions. In Science, and more specifically in Physics, the debate is currently intense around a “Theory of Everything,” which would unify electromagnetism, strong and weak nuclear forces, and gravity. Currently, among the contenders we have string theory, M-theory, loop quantum gravity and asymptotically safe gravity. There are many more. While the debate has lasted very long and is still far from resolved, physicists show us a path. There can be multiple theories. Theories do not have to be variations of each other. They can take very different angles, a little bit like crystal cleavages in crystallography. Try to cut a crystal alongside a line which is not a cleave, you’ll have a hard time. Find the right cleave and the task is easy. But there are multiple cleavages, not just one. The same goes for abstraction.

The same goes for abstraction. We can possess an imperfect theory at one point, identify that it has shortcomings, but still be unable to “fix it”. There might be a much better abstraction, but it does not come out in a trivial manner. Realizing that abstraction does not emerge from refactoring is just the start to embrace the idea that multiple viewpoints can be taken in front of the same problem.

To find good abstractions, you must find a good angle. Be flexible. Welcome input and cognitive diversity. A striking and fascinating lesson from the Bourbaki collective experience is that if you select some of the best mathematicians in the world, you still end up with an incredible diversity of profiles and techniques to resolve challenges. Teamplay works. You can have slow and deep personalities, while others are quick and shallow. The output is the same, but making the two populations work together is not easy, because their differences create distrust and even contempt.

If you succeed at not being mesmerized by an abstraction and keep a flexible mind, you must also accept that you will have to throw away some code. If an abstraction is bad, whatever the cost, reengineering must happen. It seems like an extreme statement; reengineering is costly, and projects are not always able to afford it apparently. But that’s forgetting that abstractions are to be reused, that’s their “raison d’être.” If they are deficient, the entire solution will suffer, and accepting deficiencies will cost much more than trashing a failed piece of code. Do not hesitate to write-off, even if it is painful. Realizing that you need to write off an abstraction is substantially more painful than uncovering a bug. Yet it needs to be done.

Manage abstractions, not code

“When a wise man points at the moon, the imbecile examines the finger.” This saying aptly describes the situation of abstractions and code. Peter Naur was saying that people mistake programming with the production of texts. The saying above is a little more precise with respect

to abstraction. Most organizations neglect abstractions in their management processes and pay only attention to code.

Leslie Lamport said: “People confuse programming with coding. Coding is to programming what typing is to writing. [...] programs are built on ideas.”

We find in this quote the same word “ideas” that Bjarne Stroustrup uses too. Lamport wants to carry the same message as Naur’s. Focus on ideas, not code.

Have you ever heard of an “abstractions review”. Or of tools performing “abstraction quality check”? Of abstraction “cyclomatic complexity”? Of “abstraction coverage”? “Abstraction churn”? I bet you haven’t.

Having constant debates about whether abstractions are correct, general enough, frugal enough, missing or difficult to explain or badly named should be priority number one. This is relegated to informal discussions today. Missing this is how empires collapse, how your efforts are annihilated by a competitor who had more insight.

The distinction between managing abstractions and code has powerful consequences on talent management. It is quite all right to have software developers who are more Marcel Grossmann-like than Einstein-like. Both profiles are badly needed. But the second category must not be neglected. It is my experience that the first category has a harder time dealing with the second than conversely. It is also more skeptical on average of the benefits of abstraction. It is managerial responsibility to ensure this diversity exists and the fact that the two groups operate in harmony. To the first group insisting on controlling or eliminating the second, we can quote Einstein:

> “If a cluttered desk is a sign of a cluttered mind, of what, then, is an empty desk a sign?”

An interesting final remark is on generic programming and its various implementations. When generics are instantiated through a specialization process (something that the Rust community calls “monomorphization”), like in C++ (rather than type erasure like in Java), it is generally impossible to reverse engineer and rebuild the abstractions present in the source code from the binary a compiler produces. The abstractions present in the code valuable information that the binary does not have, showing that using the word “code”, inherited from the low level “microcode” and “machine code” is an imperfect depiction of the texts produced by software developers.

Don’t get distracted

The mathematician René Thom used to say: “What limits the true is not the false, but the insignificant.”

This quote powerfully applies to software abstractions. Programming is hard, it entails creating abstractions, and abstractions are multifaceted, highly constrained brand new artifacts which require ingenuity and are intellectually challenging to produce. They are not the result of a lame copy-paste-adjust process, which is slowly decaying the quality of a piece of software.

There are lots of new third party abstractions being regularly made available. Some are the subjects of mass movements, legitimate or not. The temptation is high to follow suit. It is a variant of the old saying “Nobody has ever been fired for selecting IBM.”

Yet, you need to do your own due diligence and check multiple things, among which is the application domain, as all good abstractions come with one; you must check whether yours is a match, and you must verify whether the third-party products obey all the facets of a good abstraction and are not merely trinkets that fascinate the crowds. Remember CORBA or JavaStations? The global hysteria lasted a bit to die completely after.

The first check is the one which is usually failing. Everybody is adopting this new thing, I’ll put it everywhere. Good recent examples of this are containers (virtualization), Kubernetes and Kafka. While it is not the purpose of this document to discuss the merits of each of these technologies, beyond their intrinsic capabilities, they have been part of the fad just described. Whether you needed them or not, they were implemented everywhere without much thought, leading sometimes to painfully removing them after costly implementations, because the too eager adopters had realized they had fallen victim to mass hysteria.

Abstraction requires a lot of intellectual thinking whether you craft it yourself or you adopt it. There is no bypassing this. Witnessing industry moves should just trigger curiosity, but not an automatic irresponsible response to follow a trend and hope to participate into some name-dropping buzzword bingo and benefit from the associated wins.

It is sad to notice that many so-called software professionals are actually spending their entire time following agitations without much actual thinking beyond keeping their knowledge current on every meme they see passing by, acting like “filled bottles” rather than “fires lit” to reuse a metaphor invented by Albert Einstein. They might even hold contempt for the ones crazy enough to dare making their own abstractions.

Natural language sentences are not software abstractions

In 1978, Edsger Dijkstra published an article entitled “On the foolishness of natural language programming.” Here are a few excerpts:

> “In order to make machines significantly easier to use, it has been proposed (to try) to design machines that we could instruct in our native tongues. [...] It sounds sensible

> provided you blame the obligation to use a formal symbolism as the source of your difficulties. But is the argument valid? I doubt it.
>
> [...] Greek mathematics got stuck because it remained a verbal, pictorial activity, Moslem "algebra", after a timid attempt at symbolism, died when it returned to the rhetoric style [...]
>
> The virtue of formal texts is that their manipulations, in order to be legitimate, need to satisfy only a few simple rules; they are, when you come to think of it, an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid.
>
> Instead of regarding the obligation to use formal symbols as a burden, we should regard the convenience of using them as a privilege: thanks to them, school children can learn to do what in earlier days only genius could achieve. [...] When all is said and told, the "naturalness" with which we use our native tongues boils down to the ease with which we can use them for making statements the nonsense of which is not obvious."

During the first half of the twentieth century, the Bourbaki collective had indeed spent an inordinate amount of time standardizing and formalizing the language of mathematics. It did not invent most of the symbols, but it greatly contributed to creating the modern "code" mathematicians handle today. As an illustration, here is the definition of a continuous function on real numbers. Function f is simply continuous on I at the point a if and only if:

$$\forall x \in I \quad \forall \varepsilon > 0 \quad \exists \eta > 0 \quad (|x - a| < \eta \Rightarrow |f(x) - f(a)| < \varepsilon)\,.$$

We cannot see much of natural language in this. If ε does not depend on a, and is chosen *a priori*, it profoundly changes the definition. It becomes the definition of uniform continuity. This level of precision is essential. It is no less essential with software, possibly even more so. It is easy to turn a formal math "statement" into natural language, the other way around far less so.

As pointed out by Dijkstra, code can be manipulated using mechanical methods. The definition of a continuous function above can easily be turned into the definition of a non continuous function by a simple manipulation of the code above. Quantifiers like for all ∀ and there exists ∃, when inverted respectively yield ∃ and ∀, etc. This virtue is powerfully exemplified by the technique consisting in turning geometrical problems into code to solve them without any geometrical reasoning (whatever that means), through pure code manipulation.

Math "code" is a descendant of natural language, and the same for software. Code is the accurate, unambiguous successor of natural language.

So why do people object to this? This has to do with the fact that language is an abstraction, but an inefficient one. This has also to do with the illusion that the process of producing software is the result of the separation of thinking and doing, with some kind of user stories thrown over a wall and received by a software “coder” who executes the vision communicated to them. It completely ignores the questioning which is the unavoidable and essential part in elaborating the solution.

The cause of that can be found in the desire to see software production as a process similar to manufacturing. It is not the case. Contrary to what the eCommerce talent shock led a part of the software industry to believe, the need for thinking to produce software hasn’t decreased. It has never been a goal of abstractions. They only reduce toil. And the elasticity of the pool of talent capable of producing software has never existed in spite of all the sacrifices silently made during the last 25 years to artificially create it.

This illusion is especially powerful when large language models (LLMs) have the ambition to insert pieces of text code, or even to write entire applications. LLMs are coded to be storytellers, in the Joseph Campbell sense. Programming being abstraction crafting contradicts the intrinsic rehashing that Joseph Cambell described in 1949 in “The Hero with a Thousand Faces.” While rehashing is a positive thing when telling stories, abstraction is the exact opposite of rehashing. Its purpose is precisely to eliminate it.

The benefits of abstraction

A benefit should be understood relative to a base case. In this instance the benefit is compared essentially to abstraction-less code where constructs allowing reuse like types and functions, when they are present, represent only the result of factorization or mechanical work on raw code, without any underlying theory.

So what do we gain with abstraction? The answer is that it is better than the silver bullet or the panacea of software, to the point it can be equated to software engineering itself:

- You gain productivity multiple ways.
- You can engineer applications which are beyond the realistic capabilities of a more manual approach.
- You get closer to “code is documentation is specification.”
- Maintenance of applications is easier.
- Software is safer.

But there are precious additional benefits which are less obvious at first glance:

- Provided you use a zero-overhead programming language, performance is better than full handwritten code.
- Attracting, retaining and growing software development talent is greatly facilitated.
- Organizing people is easier.

Not using abstraction, conversely, reduces all these points drastically.

Let's take each of these points separately. They are all tautological when you consider them, to the point that disregarding them is futile. The fact that they are so obvious and so defining of software itself shows that software engineering and abstraction crafting are the exact same thing.

Enhance productivity

The fact that writing code on top of multiple abstractions saves time should be obvious. You don't have to write the code of the abstraction you rely upon. Also, writing code to form a new abstraction saves future time, because it becomes reusable, and good abstractions allow the creation of abstractions on top of them.

Depending on the abstraction height at which you are operating, the gains can be monumental, easily 10, 100 or 1,000x. It can even be more than that when you adopt an entire product doing what you need. That is the obvious maximal abstraction. But when you have to engineer a piece of software, the gains procured by abstraction are way more than what any other tool promises.

Pushing the limits of doable

This is a direct consequence of the former. The more you use abstraction, the more you are able to achieve. Abstractions are just better tools. Better tools allow us to do things that were impossible without them. The metal lathe, for instance, is one of the often neglected triggers of the industrial revolution. The same goes with abstraction.

Code is more readable

Assuming that the abstractions a piece of code relies upon are good, their usage is simple and what they do is easy to understand. Given that thanks to them, the code using them is more concise, it comes as obvious that it is therefore more readable.

Maintainability is improved

As abstractions capture what Naur calls "theory," there is no need to trash the code when part or all the team taking care of it leaves, contrary to his suggestion. The roles played by the

abstractions should be easy to understand and should have led to code easy to read, thanks to the previous point.

In most cases, as abstractions are robust, updates should be done in simple and concise code easy to read and therefore to update.

In some situations, evolving the abstractions might require revising the “theory,” which can be painful. However, reasoning over abstractions is far easier than reasoning over messy code and facing refactoring.

Two examples of code stability, once good abstractions are identified are Unix core utilities which have remained essentially the same throughout the years, and the X-Window System, which is still at version 11, released September 15, 1987 (with evolutions provided by the alternate Wayland project).

Imagine we never had Unix core utilities, or we deconstructed them, and we had to paste variations of their internals in each of our scripts. Factorization would not help without higher-order work to identify them. Bugs would be widespread and fixes would have to be applied multiple times, and we would constantly touch up the subtle variations of their ad-hoc “emanations” everywhere we use them. Maintenance would be hell. Abstraction brings order and stability in this chaos, and captures a single theory over all the variations.

Safety is improved

Every abstraction minds its own business. When you use an abstraction, you assume that it does its job properly, and that it has been tested. The very fact that abstraction is reused guarantees that it has been tested in various different ways. As an abstraction is a theory, its various test cases should be easier to inventory and implement. Reusability is one of the best guarantees for safety. You can concentrate on the new code only.

An interesting consequence of this is that if an abstraction can take risks (provided that the programming language it uses allows for it), and does take these risks, typically in exchange for performance, the risks are sealed in. The abstraction guarantees that it behaves properly according to specification. This is why the biggest contributor to safety is abstraction, not the properties of the underlying programming language. This is also why, contrary to general perception, it is better for safety to use abstraction in assembler or C than no abstraction in C++ or Rust. Alexander Stepanov summarizes this by saying “You have to know what you are doing.”

As an example of this, you can use pointer arithmetics within the code of a well-written well-tested abstraction, while the users of this abstraction do not have to take the same risks. Similarly, if you use a smart pointer, you delegate the usage of the risky memory deallocation to it, avoiding the risk of calling explicitly memory deallocation twice, or never.

An illustration of the superiority of abstraction is given by a project the cURL team was commissioned to do, offering a new back end to cURL/libcurl which allows to replace the integrated C programming language-based HTTP support with a Rust-based support. Eventually, after developing an abstraction API and some glue code to bridge cURL with Hyper, the Rust HTTP solution, the cURL team came to realize that nobody was interested. They decommissioned it with the approval of their sponsor. “100% pure Rust” and the appeal of some hypothetical gain in safety attracted nobody. Curl’s “unsafe” implementation, sealed in an abstraction inside of cURL and thoroughly tested, was preferable to all.

Bonus: Performance is better

This one is far less obvious, yet completely true. The closer you get to the limits of what you can do without abstraction, the more shortcuts you have to take to keep the maintainability of your code. These shortcuts more often than not, cause unavoidable inefficiencies.

Assuming you are using a programming language which is offering the expression of zero-overhead abstraction, supplying a high stack of abstractions to its compiler unleashes powerful optimization mechanisms which are beyond the capabilities of a human programmer. The compiler collapses the abstractions to produce an executable solution which is way too dense for a human to match with un-abstracted code. The compiler will do a much better job than a human.

The result? A surprising boost in performance, without any tradeoff on expressiveness, maintainability and productivity. It is of course difficult to quantify, but it can be as high as 100x in some situations.

Bonus: Talent management is easier

When evolving or reviving a solution, abstraction allows to reduce the reliance on a continuous team, by moving a significant part of the understanding of the theory into the artifacts.

Beyond that, being able to focus on creativity and not on repetitive tasks allows to attract, retain and grow talented software developers who are capable of leveraging the capabilities of abstraction. Good engineers want to build, abstraction offers them the tools to do it effectively.

Bonus: People’s organization is easier

Abstraction is made for reuse. Reuse creates a customer/supplier responsibility. The customer uses the abstraction while the supplier maintains the abstraction.

It is very natural to organize people around abstraction, customer code is maintained by a customer organization owning it, while supplier code is maintained by a supplier organization.

If you are constantly facing conflicts between people updating the same source code, and you are spending a lot of effort at reconciliations, you are probably not using enough abstraction.

Abstraction is progress, progress is abstraction

Progress is movement towards a perceived refined, improved, or otherwise desired state. Given the advantages we listed for abstraction, in the field of software engineering, abstraction is literally progress. The simple fact that abstraction relies on other abstractions and unleashes higher-level abstractions shows that abstraction and progress are the same thing.
When a new abstraction is created, it is created only because it did not yet exist. It is pure ingenuity. As software is eating the world, software itself is a powerful contributor to human progress.

As a powerful conclusion, abstraction in software is itself one of the contributors to overall human progress. Abstraction shapes the future. It embodies a big part of human ingenuity.

So why all the hate?

Software abstraction crafting requires ingenuity. But that's not all, the word "software" is misleading. There is no baby resting on a warm motherly bosom. "Soft" is used in contrast to "hard" as in "hardware." In fact, software has nothing soft. It is unforgiving. It is just like António de Andrada's journey. Programming involves not only a creative process but also an intense intellectual one. Being at the intersection of these two demanding human endeavors causes capable talent to be in short supply. As you can almost never rest in mechanical tasks, less capable people experience intense anxiety. This was identified very early by Edsger W. Dijkstra (1930-2002):

> Don't blame me for the fact that competent programming, as I view it as an intellectual possibility, will be too difficult for "the average programmer" – you must not fall into the trap of rejecting a surgical technique because it is beyond the capabilities of the barber in his shop around the corner.

Until a bit before the year 2000, these statements would not have been shocking. They would not have caused any irritation or even disagreement.

Around 2000, the word abstraction, the only real source of progress in software, was gone from the vocabulary of software practitioners. To this date, the word is almost taboo. Let's go back in time to understand what happened.

In 1975, Bill Gates and Paul Allen developed a BASIC interpreter for the newly-released Altair 8800. The rest is history, Microsoft was born. In 1976, Steve Wozniak developed a BASIC interpreter for the Apple I, a machine he conceived and built in Steve Jobs' parents' garage. He says that this BASIC interpreter is the most intellectually challenging task he ever did, harder than the Apple I hardware. The Apple II, and then the Commodore VIC-20, both hobbyist machines, would also have their own BASIC interpreter, a derivative from Microsoft's BASIC. Programming suddenly hit the streets.

BASIC quickly became the most popular programming language of all time. Yet, everybody knew it was not software progress for real-life applications. Everybody realized that future software would not be built using BASIC, BASIC was for tinkerers and enthusiasts. I was one of them. Games done on the same machines BASIC supported were ample proof of that, they were done in assembler.

Popularity and progress in the 1970s, 1980s and the first half of the 1990s were clearly two different things. Alan Turing in 1936 had made the point that the very purpose of software is to automate repetitive tasks, including programming ones. Software is reflexive, it eats up the world, including its own. It's Saturn eating his own children. Call this dichotomy between the "automatize-rs" people and the automatized elitist, it won't change a thing. It's what software is meant to do.

Progress in software is therefore by definition antagonistic to popularity. It pulls in the opposite direction. While children don't want to be eaten, Saturn will eventually eat them anyway. Software is meant to automate and eliminate the reliance on the human wave, the *populus*. On top, as early as 1975, in the "The Mythical Man-Month", Frederick Brooks denounced the idea of throwing more people at a failing project: it just makes it worse.

Grace Hopper in the 1970s and 1980s insisted on the fact that progress was too slow in business applications. Still, she underlined that progress was well understood, cheaper, better faster were the unforgiving metrics. Less *populus* to do the same task.

Software vendors at the same period needed to sell their software in numbers, and their incentives were to find the largest pools of users. They were tempted by presenting progress as the same as popularity. They didn't. They knew nobody would believe them if they sold the two as interchangeable. Everybody was painfully aware that software is like Saturn eating his own children.

Although the problem of developers' shortage has only one sustainable solution, abstraction, it can be somewhat mitigated by education, but the word "talent" itself conveys the fact that the only possibility to expand the pool of practitioners is identifying a larger pool of people who didn't know they had the capability and the inclination. By 2000, the attractiveness of software development jobs was already very high, the potential for expanding the pool of talent then was already quite limited.

Well before 2000, we could hear some voices asking "When will it become easier?", "Is it over yet?". Managers were starting to struggle with the idea that the doers were quickly becoming the major thinkers. Hiring was hard, it was more and more a candidate market, not a job advertiser market. The pool of talent was growing only slowly, at the pace of what the education system could produce. The industry coped with that, gently throttling the output based on talent supply and the slow pace of progress. But Alan Kay warned us that technology was shifting to pop culture.

Then, one single unanticipated event happened that shook the entire industry, especially the one around business applications. eCommerce started. It was a little bit before 2000. The opportunity was so substantial that it required millions of additional developers overnight. They needed to be created out of thin air. The planet could not supply. Technical progress fueled by abstraction was not ready, as theories do not arise on demand. Abstractions were not available to offer a bigger lever to the existing talent. For the first time ever in our history, the slow increase in available programmers became insufficient. Discreet throttling was not possible any longer. The opportunity gates were wide open, the eCommerce Gold Rush had started.

In such a situation, the only solution was the unthinkable, something akin to sabotage. Making software developers out of thin air. Sell them the notion that non-anxiogenic slop is not just acceptable, but it constitutes progress. Pitch the impossible story that people want to believe.

Surrender the well-established principle that engineering is based on sheer honesty. Accepting slop comes with substantial sacrifices. Removing the word abstraction from the vocabulary was just a part of it.

New developers needed to hear that they embraced a positive move. Something needed to replace abstraction to create that perception. Popularity was the easiest candidate. Popularity is an easy path, it’s not adversarial, and the primate mind in us finds comfort in imitation. Better, by construction, it snowballs.

Well after the eCommerce talent shock, Peter Thiel had told us, progress in software is only found by the answer to the question "Tell me something almost everybody else disagrees upon." This conveys well that unpopularity is necessary for progress to be born. We'd better listen, because whatever our agitations, our fads, our appetite to garner through vanity some personal popularity by association, our laziness when hiring or around the intellectual challenges, progress can't be stopped. As Ivan Sutherland underlined it, courage in technology is crucial. Lying to oneself and living in a world of delusions is the most powerful source for bubbles. One such has already lasted 25 years. This is so long that there is no grasp any longer of how technology could be if the normal course of progress through abstraction had been undertaken. Only a return to first principles can unveil that.

At the same time the eCommerce talent shock happened, there was still a global lack of understanding of the nature of software engineering. It was, and sadly still is, perceived as a manufacturing-like activity, where thinking and doing are essentially separate. Mentofacturing was in full swing in Silicon Valley, but the remainder of the world, especially around business applications, the least attractive domain for software engineers, was cut off from that approach and operated under the worst possible principles to produce software.

Right before 2000, it was already clear that with the progress around strongly-typed object-oriented and generic programming, crafting new abstractions was becoming an increasingly more difficult intellectual task. The surprising economics of this was and still is that thanks to the benefits of abstraction, with a reduced number of talented developers, you could achieve an overall monumental gain in productivity, quality and application features. Somewhat counter intuitive, but still true.

But how could all the sacrifices be made without anybody noticing? Sour grapes are the answer. They are easily accepted, because the ones hearing them want to believe them and will repeat them. Sour grapes turn into propaganda then dogma. Sour grapes are effective when a situation is anxiogenic, and obviously, software engineering can often be. Sour grapes are not driven by logic, it’s a world driven by feelings. We are going to see that contradiction is not an issue. It even pushes people to say that whatever they feel is “demonstrated.” And it worked.

What was the substance of this make believe?

- X is the past, Y is the future
- Machines are free
- Just throw stuff at the problem
- Response time is irrelevant
- Safety is granted by the programming language
- Bugs are features
- Less and less thinking is required
- 100% pure X
- Programming is over

And it lasted to this day as a collective hysteria. So, of course, in this context, abstraction became a curse word. Let's take each item above one by one.

X is the past, Y is the future

At the dawn of eCommerce, we started hearing "X" is the future of "Y", disguising the painful reality that "X allows to create more developers than Y while making huge sacrifices." Sacrifices were brushed away with astonishing propaganda: "machines are free" is probably the most shocking. Using the all-too-human sour grapes reaction to increasing complexity that progress dictates to turn it into a dogma. Easier is the future. Cost does not exist. This turned out to become a gigantic opportunity for cloud vendors.

We have heard it all:

- "Java is the future, C++ is the past,"
- "Python is the future, Java is the past,"
- "Full-stack [understand JavaScript everywhere] is the future, Java is the past,"
- "Vibe coding [or natural language] is the future, hand-coded JavaScript full-stack is the past."

Eric Schmidt believed so much in the story that when he joined Google from Sun Microsystems who had invented Java, that in a bout of unbridled enthusiasm, he played the missionary and wanted to convert the company from C++ to Java. Dunning-Kruger by the book, he used to be a developer. He has enough remnants of his experience as a developer to believe he is right, and not enough to see that he is wrong. Here is what happened instead, from his own words:

> You want somebody who's quick. One of the things you say is, "hire generalists." So we—in the form of Larry and Sergey to start with—had very strict rules. Here's a typical example. I would go in and I would say "Hmm, we need some experience in this layer, in this area, and we need some people who can program in Java and SQL and various other sort of XML protocols, which I was very familiar with in my previous jobs" and they

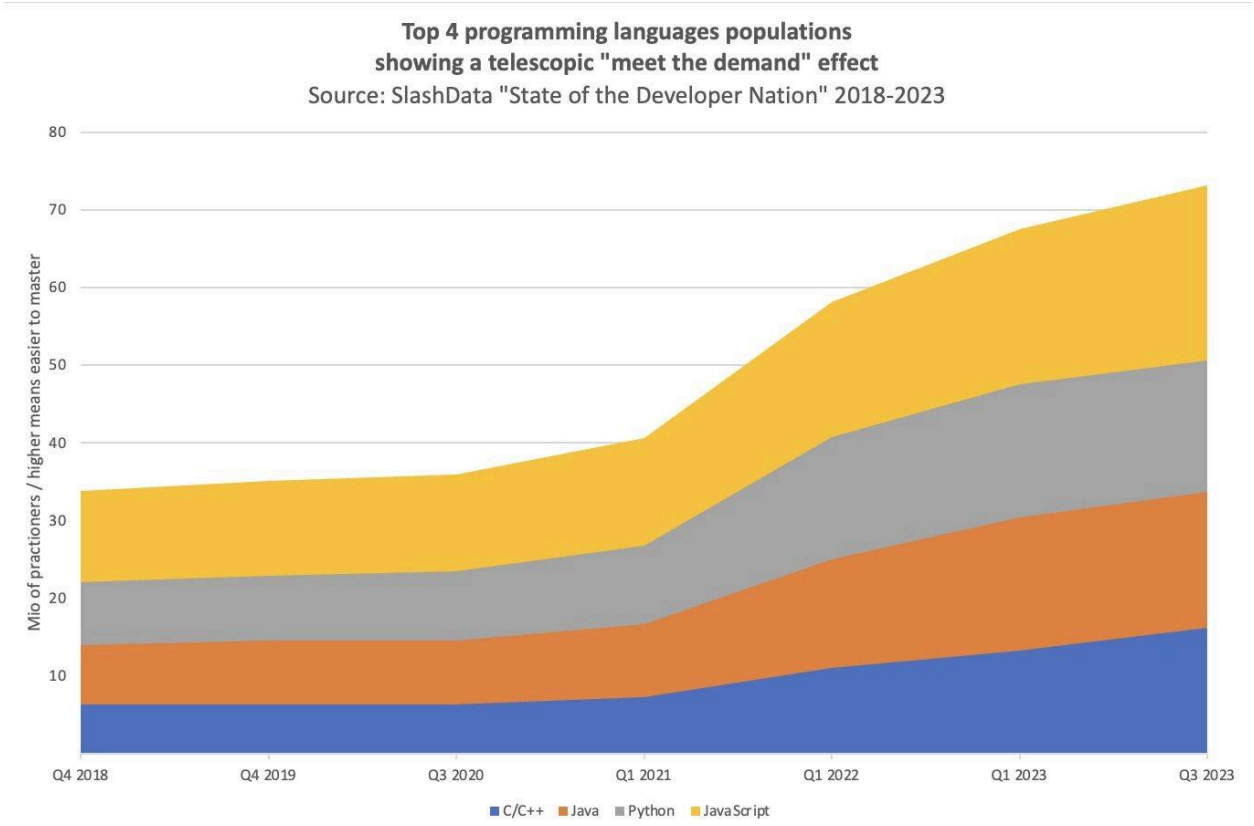
> would say “That's the stupidest thing we've ever heard. No one would ever want to use Java or any of those kinds of things." They did this partly just to annoy me. And I said “Well, who do you want to hire?" They said “We want to hire incredibly intelligent people." Well, so do I, okay. You are implying that the people working on the stuff I care about are not very intelligent. And they would say, “No, no no. We want to hire incredibly intelligent people. Incredibly intelligent people would figure out that the stuff you're talking about is stupid, and therefore, you should work on the right thing." They were actually making a different point. They were making a point that you hire people who can get the job done, that the industry over-values experience and under-values strategic and intellectual flexibility. [...] And I feel very strongly about that.

You can sense the bitterness in his joking tone when he tells the story. In the interview he feels compelled to insist that Google has Java developers now, to show that he was not that wrong. Many years later, Google still hires more C++ developers than Java ones and produces tens of millions of lines of C++ code every month. Had he prevailed, he would have killed Google. He is now unsurprisingly into AI-assisted coding. Fortunately he focused on business matters at Google and turned it into the behemoth it became, deserving everybody’s respect and applause.

To my friends who embraced the mass move to the “C++ to Java” propaganda and recited the dogma, I predicted exactly one of the next steps, “Java is the past, JavaScript is the future.” They were in denial. They said "It won't happen." Look at where we are now... It was easy on my part, is C++ too hard? Java. More and cheaper programmers. Java is strongly-typed OOP and still quite hard? Drop types, too hard: JavaScript. More and cheaper programmers. Still not enough people to fuel the human wave? Vibe coding: an infinite amount of virtual programmers for the price of a subscription. See the pattern? Each move accompanied with its own set of myths, pushed a little bit further to distract from the sacrifices, pushed under the rug. My prediction did not require much inspiration, just honesty. The last step to vibe coding is the ultimate, but all-too-predictable small iteration of creating a developers supply with infinite elasticity where it does not exist. If you found yourself surprised at the iteration that followed the one you had settled in, it means you were all too eager to embrace some anxiolytic groupthink.

Do the following experiment, take a few Java developers and ask them to estimate the ratio between the total number of Java developers in the world vs. the total number of C++ developers. It is most likely you’ll get an answer between 50x and 1,000x. I did it multiple times, it never failed. Not even once. The next reaction once you say that it’s about 1.6x is to doubt the figure, not the dogma, because people want to believe the myth. It is an easy sale.

It just happens that we can confront these mottos with independent statistics gathered by SlashData. They paint a completely different picture. To the horror of the ones embracing the mantra that popularity is the new progress, and that easy is the universal answer, we see in the programming languages landscape that since the eCommerce talent shock, everything has grown. No "X" has been the future of "Y". It was all a lie. What we see is the Boyle-Marriott law of ideal gases. The condensation of phases of matter, not the adaptation of a tool to a problem.



The number of developers for each of the top 4 programming languages is a pure consequence of the difficulty to use each of them: C++ is the hardest, no garbage collector, the largest number of keywords, the most complex object-oriented and generic programming model. Java is slightly easier, with a simplified OOP model, a garbage collector, no objects on the stack, no destructor called synchronously. Python and JavaScript follow the same trend, with JavaScript culminating with no type system to speak of.

An interesting fact about SlashData is that in the same report containing the population sizes above, they offer popularity data on which programming language is popular for what application. Except in the case of JavaScript inside browsers, which is a quasi passage obligé, all programming languages are Turing-complete and can be used for any purpose. With these hints, they contribute to perpetuating the myth that popularity is progress. The aptitude to capture as much zero-overhead abstraction as possible should be the obvious benchmark instead, because it is just a matter of time until abstraction wins.

The need for developers is fulfilled by whatever sacrifices we can make. Safety? Performance? Running cost? Maintainability? All this is an old way of thinking. Turn all organizations into manufacturing, the worst organization possible to deliver software, and voilà!

Machines are free

Performance requires constant attention, and when an abstraction sits low in the stack, inefficiencies in it cost a lot. This is tough and can be painful and exhausting, in a word anxiogenic, especially if you do not actually love programming. Sour grapes can be very potent around performance. On the other hand, you see the best developers make small changes to code and performance improvements can be massive.

If you have to create additional software developers out of a vacuum, you have to make the corresponding performance sacrifices. And by sacrifices I mean many different things, machine footprint, including RAM and processors and the number of machines required to perform a task. The “Machines are free” motto is a way to cope.

An example which is unpopular is given by garbage collecting techniques. They are powerful anxiolytics. Memory-related issues are one of the many nightmares that developers face. When using manual memory management, you need to be mindful of avoiding memory leaks, of not doing multiple releases of the same memory, of not releasing memory which was never allocated, and of not accessing released memory.

The cost of this? It is substantial and widespread, in terms of excessive RAM and CPU time consumed, but also in terms of occasional response time lag. This holds especially true as the garbage collector sits very low at the bottom of the stack of abstractions you define. Everything uses it. Most of the time what the GC does is not understood by programmers. If they did, they would probably be horrified. Not only that but garbage collectors are not the complete solution to the problem, you can still leak memory allocations that can still be reached by globals. A lot of back ends using garbage collected programming languages which were supposed to resolve the memory leak issues once and for all need to be restarted regularly to cope with that.

Without going into technical details of garbage collection techniques, suffice to say that once people are “bought into” an anxiolytic garbage collected programming language, if they are faced with a mild operational pressure, they are immediately taught to bypass it. What is a small pill to swallow after you just swallowed the first one?

The technique is not recent, it dates back at least to Lisp. Once people had started investing into the marvels of garbage collected Lisp, and faced the hardships of reality, they were told to apply the “no cons” pattern. “No cons” means avoiding allocating new memory on the heap. Of course, if you do not allocate memory, you do not have to collect it.

The “two small pills” solution to the “one large hard-to-swallow pill” has survived to this date. Once people have used enough of Java, C#, Go, or any garbage-collected programming language, they are taught to “pool objects” and manage memory by hand. Any serious programmer who has studied malloc and free knows the complexity these two simple calls entail. The hardships faced by the people who implemented them are terrifying. They must

avoid memory fragmentation, implement cell coalescence, and these are only two of the sophisticated algorithms used by malloc and free. Object pooling is naive and catastrophic. Finding excuses is so potent that in spite of the evidence of the above, garbage collector advocates won't budge. It's a perfect solution, it's only that you should not use it.

We find a similar denial with code using databases, typically in back end servers. The situation on "machines are free" is so desperate that I was talking to the head of performance at a famous database company, and he was telling me "my job is to do so that in spite of the horrors stacked up on top of our databases, applications still perform in a relatively acceptable manner." This is the reason why the surprising contempt for their own users is so widespread among database engineers.

People in desperate need of peace of mind will tell the 20/80 myth. It is a very persistent one. It goes like this "20% of the time in the back end is spent in the application, 80% in the database, so who cares about what we do in the 20%, we can do anything we want." Most of the people who repeat this legend today were not programming when it stopped being true, when eCommerce started. Opening up applications to the world implied sacrificing many of the relational database principles, the most significant of which is normalization. This means that application code runs much more of any given user-triggered service than it used to when the 20/80 rule was still valid. Since eCommerce, the ratio with relational databases has been approximately 50/50. But that's not all, being sloppy on the application side has caused a situation where more efficient and modern database solutions like NoSQL/document databases have exhibited a ratio even more stressful for application code, and we have inverted the 20/80 rule. The 20/80 rule has become an urban legend justifying the slop, with enough people finding comfort in repeating it, it becomes true.

On the same topic of performance, realize that serious programming languages creators are like the database performance person above. They care about the smallest details to ensure that every drop of performance is extracted from the code. C++, Rust, Swift and C# (but not Java) all implement for instance some kind of specialization (the Rust community calls it "monomorphization") when they compile generic code. All this is pointless when people produce irresponsible code.

No surprise that nobody is talking of Green IT. "Machines are free" has created a situation where nobody knows how frugal their solutions could be, would they care a little. Can you imagine what 100x would mean for the planet?

Just throw stuff at the problem

It was the 1980s, the dawn of mainstream object-oriented programming. The industry was starting to look at OOP as the next big thing. It wasn't limited any longer to labs, research, visionaries, hobbyists and tinkerers. Structured programming was tired, OOP was wired. It was very clear that OOP was here to stay, but the enthusiasm around it was way overboard.

I participated in a vast European project with a multitude of private companies making their contributions and working together. That ended up being the only positive outcome, but that's another story.

With such diverse participants, each with their own agenda, even technical working sessions were very animated. These were times when, like today, the separation between thinking and doing was in full swing. Hands-off technical managers were already a thing, and they insisted on participating. Why not?

Of course, hands-off managers had no practical experience of OOP. They had read articles in the press, and it was fashionable to recite all the merits of that (not so) brand new approach. In every meeting with high technical stakes one of the managers used to cut everybody and exclaim "It's an object!" This interjection disrupted discussions and people who expected an answer had a hard time debating any topic in depth. "It's an object!" was the final answer to everything.

Many years later, nothing has changed, mass movements are even bigger and easy answers have never had so much appeal. "It's Kubernetes!", "It's Kafka!" Whether you need something as heavy as Kubernetes or not, whether you need a queueing system or a streaming system, Kafka has to be present. It's AI! It's Agentic! It’s REST! It’s WebSockets! Forget engineering professionalism and first principles, throw popular technology at the wall, see what sticks.

There are many advantages, but the biggest is that it is easy. Things getting easier in software has always been the irrational hope. Panacea, dear Panacea, please hear us! Meanwhile, cloud vendors are so happy.

Response time is irrelevant

"Nothing but a fast-loading search site", was Google advertising in 1999. Indeed, if you are old enough, or go back in time and investigate, you will know that Google did not emerge from being the largest store of web pages. It did not emerge because of the smart PageRank algorithm. It surpassed its competitors because it was simply… fast. So fast that Google was proud to display on the response page how fast it answered your search. Google cares so much that they work hard at making the landing pages and information exchange between the browser and their servers as small as possible. Google founder Larry Page's Wikipedia page says:

> Larry Page has mentioned that his musical education inspired his impatience and obsession with speed in computing. "In some sense, I feel like music training led to the high-speed legacy of Google for me". In an interview Page said that "In music, you're very cognizant of time. Time is like the primary thing" and that "If you think about it from a music point of view, if you're a percussionist, you hit something, it's got to happen in milliseconds, fractions of a second".

The idea that short response time drives user engagement is not new, it predates Larry Page. Google just has the intellectual honesty and the focus to care for it. There is even a technical definition of "instant response time". Users perceive instant response time if the product they are using takes less than 300 milliseconds or so.

When you think about it, it's not easy, because if you are running back ends and serve users on the entire planet from a single location, even with a high quality network, you will realistically face a latency of about 200-250 milliseconds, without even running a single instruction in your back end machine. What's left to do any kind of value-added computing is very slim.

Two anxiolytic solutions to this have been found and used:

- Ignore all this, claim good response time does not matter, cover your ears with your hands and sing "la la la" in a loop.
- Install multiple machines everywhere near customers to beat the speed of light by shortening distance, face nightmarish consistency issues, bury all your slow back end code under asynchronous Chernobyl sarcophagi turning application behavior into a disconcerting experience for users. Ask tons of money to fund the armies to run and maintain that and the cloud services to support it.

The good thing with the second solution is that it is consistent with the previous technique, which is to "throw stuff" at the problem. Put Kafka or popular event architectures everywhere.

Safety is granted by the programming language

This is a recent one. We have seen that the largest contribution to safety in an application is abstraction. The separation of concern allows spending much more on testing, because there is a great deal of reuse, and it establishes borderlines between callers and callees, which serve as verifiable contracts. Callees can take as many unsafe risks as their performance objectives dictate. We know that in our day to day experience, for instance, deadly capacitors are present in power adapters, instead of making them safe, we seal them inside an enclosure with a big sticker saying “don’t open it.” The cost of making the inside of a power adapter safe would obviously be both prohibitive and pointless except to serve some kind of misplaced obsession.

But in a context where abstraction is a taboo topic and the world is essentially flat, the only last resort guardrail you are left with against programmer slop is whatever the programming language can offer. The programming language becomes the tool to control programmers like cattle, if it can.

The technique to sell this to desperate people is not new. It is based on the dream of purity. When XML came out, it was not perceived as a simple, albeit expensive data presentation, it was promoted under the motto “100% pure XML.” When Java rose to prominence from its initial purpose of running in embedded systems to being the garbage collected programming language capable of feeding the eCommerce beast with millions of new developers, it followed suit: “100% pure Java” was born. 100% pure “blood” largely predated even XML and has accents which are distasteful. But it is all too human to be enthralled by it, and it sanctions any deviance to some breeding standards, pushing to sectarism against simple common sense.

Safety has followed the same trend. The Rust programming language, a very respectable creation, has been promoted using the same foul-smelling cult of purity technique. Either the programming language possesses the safety genes or it does not. If it does not, the infestation has to be eradicated. The message is relayed by zealots everywhere, while abstraction is still the taboo topic.

Maybe the purity argument has been abused, because it seems that Rust activists are reduced to spreading money around to force the materialization of the "100% pure Rust blood." Daniel Stenberg, the creator of the famous curl/libcurl product, took money from Rust activists to replace his existing C-based HTTP implementation with a Rust-based implementation. At the end, unsurprisingly, he saw that nobody was actually interested in a pure blood approach, and his C abstraction of HTTP sealing "unsafety" in, turns out to be a better solution than trying to control developers' freedom at programming language level.

Safety coming from the programming language itself is an exaggerated promise similar to garbage collection solving all memory problems. The irony is that many of the ones who swore by garbage collection and claimed it had no cost are now chanting the merits of Rust and condemning garbage collection in favor of the new-to-Rust, but very old, reference counting

technique. Obviously it is easier to switch from one propaganda to another than from one truth to a contradicting one. In any case, Rust is a sophisticated programming language, which is difficult to master, and the wish that it might create even more developers out of thin air is clearly unsubstantiated.

Resolving memory safety though mere sectional analysis of code is an obvious fallacy. The deeper problem can be described through the expression "referential integrity," which has more to do with reevaluating how heap objects point at each other, from dusty good old pointers a bit improved with polymorphic types to true relationships with automatic behaviors. This has almost nothing to do with sections of code to handle data, and everything to do with qualifying the data themselves, in true respect of the object-oriented approach legacy.

Bugs are features, it always does… something

There is nothing more humiliating than a software crash. More generally, bugs of all kinds, memory leaks or simply functional misbehaviors trigger investigations which are some of the most difficult software endeavors. On average the best engineers produce slightly fewer bugs, but more importantly, bug identification and fixing requires method, creativity, intellectual flexibility, rigor and of course a talent to update complex pieces of code. It's very rare that a bug happens in a simple location. So there is no situation that challenges the quality of engineers more than facing bugs. When trying to create engineers out of thin air, something has to yield, especially around bugs.

We have already talked about garbage collecting techniques. They are a patch to solve a lot of memory-related issues, but not all of them as we saw. Instead of fixing memory bugs, people developing server-side often choose the dirty solution of rebooting their servers from time to time, for instance daily. Bugs become a fact of life, like commensal bacteria living in the production guts. Of course the automation cost to do this properly is very substantial, and pushed under the carpet. But even a garbage collector won't protect you against rapid consumption of RAM. There is no solution for that except debugging with the proper tools. Garbage collected programming languages such as Java are still hard to master. This can be seen in the relatively modest tier of developers Java allowed to create "on top" of the C++ developers tiers. One of the hurdles is the typing system, which causes the Java compiler to be perceived as pesky. The solution to that is typeless programming languages such as JavaScript. Server-side JavaScript for instance is popular, and has become synonymous to the initially-neutral "Full stack" expression. Nowadays "full stack" means JavaScript everywhere. Once types have been removed and pesky compilation has disappeared, only memory problem can causes

Conclusion

As software is eating the world and its progress is fueled by abstraction, we can conclude that most of mankind’s progress is now linked to abstraction, which is quite a statement.

Pushing abstraction further requires using tools, programming languages for instance, which allow capturing the maximum possible amount of abstraction, carefully avoiding boilerplate as much as possible.

It is the exact opposite of the hope that thinking will gradually disappear. Good abstractions stimulate thinking, they do not remove it.

Pushing this logic as far as humanly possible, we should never complain that our tools are “too complex” if they allow us to gain every possible inch in expressing good abstraction. We should judge them precisely for that ability. By “good” I mean at least general, frugal and allowing to think in a way unleashing the ability to express creative solutions.

Credits

I wish to thank the following list of people for their time and contributions to this work. Their help is in no way an endorsement.

Ariel Otilibili Anieli
Stéphane Dalbera
Dezmin Danehy
Mathieu Eveillard
Steve Heller (https://2misses.com, variable-length-native hash table inventor, details here)
Ed Hodapp
Nicolas Maillot

This document is a live one. I intend to make it evolve, updating the version number at the beginning every time a new revision is made. I have much more material than the document shows, but doing software taught me that there comes a time when you have to let go and release a piece of work. Feel free to send me remarks, if I incorporate a suggestion of yours in this document, however small, I'll gladly list your name above. Please do not be offended if I don't accept all modifications.