Metaphors We Compute By

**CODE IS A
STORY THAT
TELLS HOW
TO SOLVE A
PARTICULAR
PROBLEM**

ALVARO VIDELA

In their now-classic book *Metaphors We Live By*,⁶ George Lakoff and Mark Johnson set out to show the linguistic and philosophical worlds that metaphor isn't just a matter of poetry and rhetorical flourish. They presented how metaphor permeates all areas of our lives, and in particular that metaphor dictates how we understand the world, how we act in it, how we live in it. They showed that our conceptual system is based on metaphors, too, but since we are not normally aware of our own conceptual system, they had to study it via a proxy: language.

IN THE BEGINNING WAS THE WORD

By studying language, Lakoff and Johnson tried to understand how metaphors work by imposing meaning in our lives. The basic example they present is the conceptual metaphor "argument is war." We understand the act of arguing with another person in the same way we understand war. This leads to the following expressions in

our daily language:

- Your claims are *indefensible*.
- He *attacked every weak point* in my argument.
- I *demolished* his argument.
- I never *won* an argument with him.

These sentences may seem innocuous, but the problem is how we act and feel based on them. We end up seeing the person we are arguing with as our *opponent*, someone who is *attacking* our positions, so we structure argument as if we were at war with the other person. This means the metaphor is not just language flourish; we live it. Lakoff and Johnson propose the exercise of imagining a culture where arguments are not viewed in terms of war—of winners and losers—but in which language is a dance, where you have to cooperate with a partner in order to achieve a desired goal, reaching conclusions as a team.

The book goes on to analyze the different aspects of language and metaphors and how they affect our concepts and view of the world. The authors give many examples to defend the thesis that our understanding of the world is based on metaphors and that those metaphors are the foundation of behavior.

One of the book's biggest takeaways is that metaphors enable certain ways of thinking, while restricting others, as the argument-as-war example illustrates. This article applies this idea to computer science. How do metaphors shape the way we understand computing and its related problems? What kinds of problems are enabled by the metaphors in use? And, no, monads are not like burritos!⁸

First, the article looks at how metaphors help us understand the relatively young world of computers

Metaphors
also
obscure
possibilities
if you
don't understand
their limitations.

and how they affect the way we structure code or design algorithms and data structures. We even solve problems based on which metaphors are part of our arsenal, or *toolbox*. “Sometimes our tools do what we tell them to. Other times, we adapt ourselves to our tools’ requirements,” states author Nicholas Carr in his book *The Shallows*.³ Metaphors are the tools of comprehension.

A METAPHORICAL UNDERSTANDING

People understand new concepts by relating them to what they already know. Back in the late '40s and early '50s when today's computers came to life, no word for such an invention existed, but people understood them as automatic brains. Actually, the word *computer* existed at that time, but it was used to refer to the person who did calculations for engineers. Think of engineers needing to know the trajectory of a projectile or how the wind would affect an airplane's wing shape; they would throw a couple of formulas and numbers to their *human computers* to get the answer. Now came these new machines that did the work automatically; they called them *electronic computers*, eventually dropping the *electronic* part of the name. So, our very own discipline was named after a metaphor.

But metaphors also obscure possibilities if you don't understand their limitations. A common problem with new metaphors is that the original meaning of the word is used at face value. The word being used to explain a new concept may actually limit understanding of that very concept. In his book *The Information*,⁵ James Gleick gives a fascinating account of the invention of the telegraph.

The word *tele-graph* means far-writing. Lo and behold, early telegraph prototypes were strange machines that tried to write at a distance, using one-to-one mapping of letters of the alphabet into wires, which sounds terribly impractical. Around that time, thanks to Louis Braille, people began to understand that language could be coded in a form different from the way it sounds (or is written). Morse code was the next step in improving the telegraph and in understanding that people don't have to "write at a distance" to have actual long-distance communications.

Thanks to mathematician Claude Shannon and others like him, we managed to escape from the problems of the *telegraph* metaphor and build the whole discipline of information theory beginning in the late 1940s. Shannon's seminal book, *The Mathematical Theory of Communication*, put forth the basic elements of communication: information must be encoded first, then sent into a channel to be decoded at the other end, so the destination receives it.⁷

METAPHORS AND CODE

A well-known unattributed quote [often misattributed to Charles Baker] is, "To program is to write to another programmer about our solution to a problem."² A program is an explanation of how a problem might be solved; it's a metaphor that stands in place of a person's understanding. For metaphors to be effective, however, they need to convey meaning using concepts already known to us. The explanatory power of a particular program can be used as a measure of its own elegance.

Consider the following example. Say you could program

By choosing the right metaphor, your program reaches a level of abstraction that requires the least amount of effort for someone foreign to the problem to understand the solution.

a computer to command other computers to perform tasks, respecting their arrival order. This description is already hard to understand. On the other hand, you could describe the solution by describing a *queue server* that assigns *jobs* to *workers* based on a *first come first served* queue discipline.

A queue is a familiar concept from daily life, seen at the supermarket, the bank, airports, and train stations. People know how they work, so for someone reading your code, it might be easier to talk about queues, workers, and jobs than trying to explain this setting without using the *queue* metaphor.

By choosing the right metaphor, your program reaches a level of abstraction that requires the least amount of effort for someone foreign to the problem to understand the solution. Also, solving the problem with queues provides a whole mathematical theory for free. Mathematics is itself a field where problems are tackled only when an appropriately expressive language is available to approach them. With the queue metaphor, you are no longer punching blindly in the dark. Now you can analyze and understand the problem with all the tools provided by queueing theory.

DATA STRUCTURES AS METAPHORS

Analyzing data structures helps us to see which would be a better fit for the performance characteristics of a particular problem, but we often forget that a data structure also contains explanatory power.

The data structure choice helps convey meaning. You could store a bunch of elements in an array, but what if

you need to ensure that the elements are unique? Isn't that what *sets* are for? The underlying representation of your set can still be backed by a plain array, but now your program expresses its intentions in a clearer way. Whenever other programmers read it, they will understand that the elements in your collection must be unique. It is important to realize that a program is just another succession of bits that a computer needs to process. It is the programmer who gives meaning to those bits, so you have to use the right metaphor on top of them to make your program clearer. As has been said, "no one has seen a program which the machine could not comprehend but which humans did."²

You must strive to make your program as easy as possible for other programmers to understand. Ease of comprehension should be the standard by which programs are measured. Keep in mind that you can arrange code in many different ways to solve a computing problem, but not all those arrangements will favor human communication and understanding. You must ask yourself, by reading my code, will others understand how I solved this particular problem?

Just as metaphors enable certain ways of understanding and limit others, so do data structures. Earlier we saw the problem with using direct metaphors such as the original telegraph. When it comes to data structures, you can see that a set will reveal that its elements are unique, and it will allow you to test if an element is a member of the set. With a linked list, however, you get the idea of traversing its elements one after the other, without being able to skip them. With an array

you get the idea that you can address its elements by index. The same can be seen with queues and stacks, two of the most fundamental data structures taught in any algorithms course. Each can be implemented using an array—the difference is that one returns the elements in FIFO (first in, first out) order, while the other does so in LIFO (last in, first out) order, respectively.

Even if this looks like an everyday thing for most programmers, the moment you choose to use a stack instead of a queue, you are deciding how to explain your program. The stack is a very good metaphor for the collection of items that a program works with, because it tells a future reader of the program in which order to expect the items to process. You don't even need to read how the stack is implemented, since you can assume you will get the items in LIFO order. This is why types are so important in computer science—not types as in static type checking of programs, but types as the *concepts* used to describe programs: persons, users, stacks, trees, nodes, you name it. Types are the characters that tell the story of your program; without types, you just have operations on streams of bytes.

COGNITIVE LEAPS

The goal is to find the right metaphor that describes and explains a problem. As explained earlier with the queueing theory example, a cognitive leap was needed to go from tasks that have to be processed in a certain order to understanding that this is a queueing problem. Once you manage to make that cognitive leap, all the mathematical tools from queueing theory are at your disposal. Graph

theory is filled with examples of mundane tasks that, once converted to a graph problem, have well-known solutions. Whenever you ask Google Maps to get you to your destination, Google translates your problem to a graph representation and suggests one or more *paths* in the graph. Graphs are the right metaphor, understood by mathematicians and computers alike. Are there other instances of problems that seem difficult but that can be solved by finding the right metaphor? The distributed-systems literature has a very interesting one.

In the late '80s Alan Demers and his colleagues from Xerox tried to find a solution to database replication in unreliable networks. They classified their algorithms as “randomized,” explaining them by using the rumor-mongering metaphor: a computer would tell two other computers about an update, then each in turn would tell two other computers about the update, and so on until the information was replicated across the system. This metaphor gave way to a new area of study called gossip algorithms. The *gossip* metaphor makes the idea easy to explain, but the Xerox team was still lacking the mathematical tools that would help analyze the effectiveness of the algorithms.

During their research, they discovered another metaphor related to their problem: epidemics. They understood that their algorithms replicated data the same way in which an epidemic disseminates across a population. By using this new metaphor, they got immediate access to all the knowledge in *The Mathematical Theory of Epidemics*,¹ which fit their work like a glove. Not only did they name their paper “Epidemic Algorithms for Replicated

Programmers must be able to tell a story with their code, explaining how they solved a particular problem. Like writers, programmers must know their metaphors.

Database Maintenance,”⁴ they also took the nomenclature of that discipline to explain their algorithms. It was a matter of finding the right metaphor to get access to a new world of explanatory power.

METAPHORS EVERYWHERE

Do we really use that many metaphors in programming? Let’s take a look at an example from the distributed-systems literature (metaphors are in *italics*):

Whenever nodes need to *agree* on a common value, a *consensus* algorithm is used to *decide* on a value. There’s usually a *leader* process that takes care of making the final decision based on the *votes* it has received from its *peers*. Nodes communicate by sending *messages* over a *channel*, which might become *congested* because of too much *traffic*. This could create an information *bottleneck*, with *queues* at each end of the *channels* backing up. These *bottlenecks* might render one or more nodes *unresponsive*, causing *network partitions*. Is the process that’s taking too long to *respond dead*? Why didn’t it acknowledge the *heartbeat* and *trigger* a timeout? This could go on, but you get the point.

A STORY IN CODE

Programmers must be able to tell a story with their code, explaining how they solved a particular problem. Like writers, programmers must know their metaphors. Many metaphors could be used to explain a concept, but you must pick the right one so your code conveys your ideas to other programmers in a precise and meaningful way. Thus, you cannot use every metaphor you know. You

must master the art of metaphor selection, of meaning amplification. You must know when to add and when to subtract. You will learn to revise and rewrite code as a writer does. Once there's nothing else to add or remove, you have finished your work. The finished program in front of your eyes is now the solution to your initial problem. Is that the meaning you intended to convey in the first place?

References

1. Bailey, N. T. J. 1957. *The Mathematical Theory of Epidemics*. Arnold.
2. Charles Baker (Ed.). 1967. What a programmer does. *Datamation* (April); http://archive.computerhistory.org/resources/text/Knuth_Don_X4100/PDF_index/k-9-pdf/k-9-u2769-1-Baker-What-Programmer-Does.pdf.
3. Carr, N. 2011. *The Shallows*. W.W. Norton.
4. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J. 1987. Epidemic algorithms for replicated database maintenance. *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*: 1-12.
5. Gleick, J. 2011. *The Information: A History, a Theory, a Flood*. Pantheon.
6. Lakoff, G., Mark J. 1980. *Metaphors We Live By*. University of Chicago Press.
7. Shannon, C. E., Weaver, W. 1949. *The Mathematical Theory of Communication*. University of Illinois Press.
8. Yorgey, B. 2009. *Abstraction, intuition, and the "monad tutorial fallacy."* <https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>

Acknowledgments

I would like to thank Jordan West and Carlos Baquero for all the discussions we had about how metaphors permeate computing, and for the feedback they gave me once the article was written.

Related articles

➡ First, Do No Harm: A Hippocratic Oath for Software Developers
Phillip A. Laplante, Penn State University
What's wrong with taking our profession a little more seriously?
<http://queue.acm.org/detail.cfm?id=1016991>

➡ Coding for the Code
Friedrich Steimann and Thomas Kühne
Can models provide the DNA for software development?
<http://queue.acm.org/detail.cfm?id=1113336>

➡ A Nice Piece of Code
George V. Neville-Neil
Colorful metaphors and properly reusing functions
<http://queue.acm.org/detail.cfm?id=2246038>

Alvaro Videla (*alvaro-videla.com*; *@old_sound*) works as a distributed-systems engineer for a Silicon Valley company out of his home in Switzerland. He was a core developer of RabbitMQ, and he was the lead developer of the team that built one of Germany's biggest dating websites. He is the co-author of the book *RabbitMQ in Action* [2012], from Manning Publishing. Copyright © 2017 held by owner/author. Publication rights licensed to ACM.