The Law of Leaky Abstractions

Synopsys and Analysis

Software Engineering 2018

Bruno Carvalho
up201606517

Diogo Yaguas
up201606165

Tiago Castro
up201606186

October 6, 2018

Synopsys

The Law of Leaky Abstractions[1] is an article written by Joel Spolsky, now CEO of StackOverflow, with the intent of demonstrating to the reader that programming abstractions are not perfect and are merely ways of making a programmer's life easier, they should not be relied on completely when aiming to be a good programmer.

To introduce the concept of **leaky abstractions**, Joel focuses on data exchange protocols using TCP as an example – a way to transmit data that is reliable and is a key piece in the engineering of the Internet which we rely on every single day – which is an abstraction based on IP – an unreliable method that doesn't guarantee the arrival of the messages or their integrity. TCP attempts to provide a complete abstraction of an underlying unreliable network, but sometimes, the network *leaks* through the abstraction and you feel the things that the abstraction can't quite protect you from. To show how this abstraction may fail, he uses the hypothesis of a "pet snake that chewed through the network cable leading to a computer, and no IP packets can get through, then TCP can't do anything about it and the message doesn't arrive". This is the point where the key phrase of the whole article is presented:

> "All non-trivial abstractions, to some degree, are leaky."

[1] Joel Spolsky. *The Law of Leaky Abstractions*. URL. Nov. 2002.

Learning how to use the basics of the base on which an abstraction is built is needed when learning and using the said abstraction. An example of this need is the use of strings in C++:

> "When I'm training someone to be a C++ programmer, it would be nice if I never had to teach them about char*'s and pointer arithmetic. It would be nice if I could go straight to STL strings. But one day they'll write the code "foo" + "bar", and truly bizarre things will happen, and then I'll have to stop and teach them all about char*'s anyway."

Summing up the whole point of the text, the law of leaky abstractions means that whenever somebody comes up with a wizzy new code-generation tool that is supposed to make us all ever-so-efficient, you should learn how to do it manually first, then use the wizzy tool to save time so that when the abstraction fails, you know how to fix it. This way, you will be a better, more complete programmer.

> "When you need to hire a programmer to do mostly VB programming, it's not good enough to hire a VB programmer, because they will get completely stuck in tar every time the VB abstraction leaks."

Analysis: Impact on Software Engineering

The analysis presented in the article can be lifted to present a more general outlook on software development, particularly the structure of programming languages. When we look at the vast territory of programming languages out there, we can classify them as low and high-level pretty easily. We do so based mostly on how complex are the abstractions that they provide for elementary and direct interactions with the hardware. These abstractions are built on top of other abstractions, from lower-level languages, forming layers of abstractions – often many of them for the highest-level languages. This is analogous to a code database written in a single language (like an operating system), a layered communication protocol, and more.

The abstractions provided by the lower level layers are inevitably leaky, as they are slaves to the hardware limitations of the machine they are running on (not to mention legacy issues, conventional disputes, etc.). An optimal, perfect high-level language wouldn't have any of these limitations, but the abstraction that is built on top of layer cannot fundamentally be rid of the limitations of the lower one - the abstraction *leaks* - and at best it can transform them into a different restraint, considered more negligible or easier to handle: demanding more or unbounded time/memory to complete some operation, demanding boilerplate and verbose code...

The developer who is oblivious to this and faces one such problem stemming upwards from a low-level layer, will attempt to solve it only at the top level, ignorant of its causes, and is doomed to fail. He discovers he needs to learn the functionality of the lower layers to find how to circumvent and solve his problem, or ultimately accept he needs to take an entirely different approach.

The law of leaky abstractions pushes the learning curves upwards, making them ever so steeper in every area of software engineering.