2024-05-02-Free Range Programming

Processes vs. Closures

Processes and closures are effectively the same, with processes being better since they use hardware (MMUs) to protect memory even better.

To me, the issue is memetic, not technical. What I seem to be wanting to say, but, keep swirling about and not saying, is that subtle, minor changes in notation have gross effects on what one is able to invent. "Language affects thinking" and all that, but, what I'm driving at seems to go even deeper.

In this instance, simply *thinking* about processes as processes instead of as closures shuts off whole avenues of thought. Processes are associated with heavy-weight constructs that are only allowed to exist in operating systems and are *soooo* inefficient that you'd have to be crazy to use them at the programming level. UNIX showed that pipelines of processes were quite useful, but, because processes were used, it seems that the idea has been dropped from common use. In fact, people seem to think that they can build pipelines using functions - false. If you simply *think* about pipelines of closures - instead of pipelines of processes - new ideas begin to flow.

I seem to have been indoctrinated with the idea of pipelines at an early age. I've been striving to make processes cheaper to use, so that we could all begin to use pipelines at the programming level. I'm not afraid to use big, heavy shell-outs using operating system processes in my designs, since I know that my designs can be optimized and processes can become as cheap as function calls, i.e. closures.

This very subtle change in thinking - lack of fear of inefficiency - allows me to dream up different ways to solve problems. If I think in terms of pipelines, I immediately jump to thinking about components which immediately suggests components with multiple inputs and multiple outputs which immediately jumps to the idea of building software using LEGO® blocks. Aside: I have concluded that you can't use functions to build LEGO® blocks, you have to use queue-based units of software. Aside to aside: of course you *can* build LEGO® blocks with functions, but the result is relatively clumsy and discourages further thinking along those lines.

Functions imply *blocking* - the caller must suspend and wait for the callee to return a value. Components - daemons, servers, statecharts, etc. - don't imply *blocking.* Fire and forget. Components are asynchronous. Functions are synchronous.

If you begin with thinking that you must use functions, then you strive to solve all problems in some synchronous manner.

If you begin with thinking that you must use components, then you are free to plug-and-play and solve problems in an asynchronous manner, and, to apply synchronization only when necessary. A network protocol is synchronization. Shaking hands with someone when you meet them is a protocol. Delaying a meeting until the CEO arrives is a protocol - you *could* proceed with the meeting, but, you *choose* to wait. This is asynchronous thinking with synchronization layered on top of it. Synchronous thinking, though, lathers a micro-protocol onto every little step in the process. You have no choice in the matter - you have to delay the meeting until the CEO arrives. At best, you wait until the CEO calls and says "go ahead, start without me" - that's synchronization, your actions are intimately tied - coupled - to the CEO's actions. Everything is Centrally controlled, you have no "free will" and no freedom to choose. Going further, if the CEO has to deal with some other important matter and forgets to call, you are left hanging.

Banning the use of If-Then-Else

"If-then-else" has been one of the banes of our existence. The concept is too low-level. To get useful control flows, you have to tie *variables* into the equation and, then, you get into the issues of global variables, free variables and those sorts of things.

On the surface, it seems that "if-then-else" is extremely useful and cannot be a fundamental problem, because we've been indoctrinated to believe in the existence of if-then-else.

If-then-else was invented to implement conditional values of functions when using digital CPUs and subroutines. That's probably why McCarthy called the programming construct COND.

If-then-else was not originally meant to implement interesting control-flows and to abstract-away the use of GOTO.

We applied band-aids to our methods of programming CPUs, instead of stepping back and fixing the underlying problem, i.e. banning the use of low-level "if-then-else". This is like dispensing Tylenol® to dull pain, while not curing the cancer.

We have applied band-aids to the "problem" of control-flow in CPUs and subroutines. For example, we declare edicts such as not allowing globals, not allowing side-effects, etc. These edicts obviously contradict Reality. Servers and daemons, of course, have side effects, but our band-aids tell us that this cannot be possible. We become mentally paralyzed by cognitive dissonance. For example, programmers think that "concurrency is hard" only because our band-aids weren't designed to accommodate concurrency, yet, 5 year-old children learn hard real-time concurrency (piano lessons, reading music) without needing PhD degrees.

What can we do about this problem? How can we replace the use of if-then-else, while still achieving useful control flows? We've already seen small solutions to the problem of if-then-else in function-based programming[1], e.g. in various *map()* functions. These are basically functional expressions of hoary bits of control flow that happen under-the-hood. We see ideas in FP creeping towards the goal with concepts like *pattern matching*.

With developments like OhmJS (based on PEG - parsing expression grammars), though, we can go whole-hog. We can invent textual syntaxes that express any control flow that we desire.

Simply looking at an OhmJS grammar reveals a control-flow that would be hard[2] to implement using if-then-else. OhmJS expresses a backtracking control-flow. "Try this branch, and, if it fails, backtrack and try the next branch...".

[1] I consider function-based programming to be a superset of the current fad of FP-based languages. Function-based programming began in the early days of computing with languages like FORTRAN and Lisp. It was deemed convenient to use CPU subroutines to fake out mathematical functions. It appears to have been forgotten that the relationship is a one-way mapping only - functions can be represented using CPU subroutines, but, CPU subroutines are not functions.

[2] confusing

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*