Divide And Conquer

When a problem looks to be complicated, a strategy for attacking the problem is “divide & conquer”.

Most programmers know about Divide & Conquer.

What problems look to be complicated?

For one – multi-tasking is considered “hard” by many programmers. Multi-tasking is actually simple, but needs to be further divided (and conquered). For example, multi-tasking, as it is know today consists of several sub-problems:

- time-sharing
- memory sharing
- networking, IPC, communication
- concurrency.

Let’s apply Divide & Conquer to the multi-tasking. (1) Time-sharing is needed only by operating systems, such as Linux, Windows, MacOS, etc. Let’s throw time-sharing aside. Gnarly problems, like priority inversion go out the window.

Memory sharing was an issue when memory was expensive. Memory is no longer expensive. Let’s throw memory-sharing aside. Thread-safety, etc., go out the window.

We are left with networking and concurrency.

Networking is easily reduced to it most basic form – a wire.

Concurrency, at its most basic form is 2 apps communicating across a wire.

We know how to write the two apps –say, using Python, JS, etc., etc..

Yet, we don’t have a “language” for app-to-app communication. There is no popular Python-for-comms language. There are budding attempts at this kind of language, the most common being UNIX® *bash*, but *bash* is tangled up in complexity – time-sharing and memory sharing and variables and, etc., etc. ... FBP[1] is a not-popular-enough attempt as this kind of language, but it tends to be tangled up with multi-tasking libraries which are tangled up with time-sharing and memory-sharing. The FBP site references Linda. CPS[2] and CSP[3] are text-only attempts to tackle this problem – they simply demonstrate that the text-only mentality does not extend well to concurrent applications. TC;DU (Too Complicated ; Didn’t Use).

An acquaintance of mine builds products that measure the health of race horses. His app uses some 37 processors. He uses a language called MicroPython[4]. He has no problem with multi-tasking. One

1 Flow-Based Programming <ref>
2 Continuation Passing Style
3 Communicating Sequential Processes <ref>
4 https://micropython.org/

processor, one thread. Node.js? Why bother? Processors are cheap. Linux? Why bother? Processors are cheap. When he wants to get *really* complicated, he creates an event loop on a processor, that checks for incoming events and does some work in the background.

We need a lean language for coordinating a hierarchy of processors[5]

We need to apply divide & conquer – throw out operating systems, throw out heavy-weight thread libraries, throw out memory sharing, etc.

Fractal Design

Divide & Conquer and Hierarchy leads to thinking about the Design task as a fractal. Every Component in a Design "does one thing well" and leaves the rest for further sub-dividing.

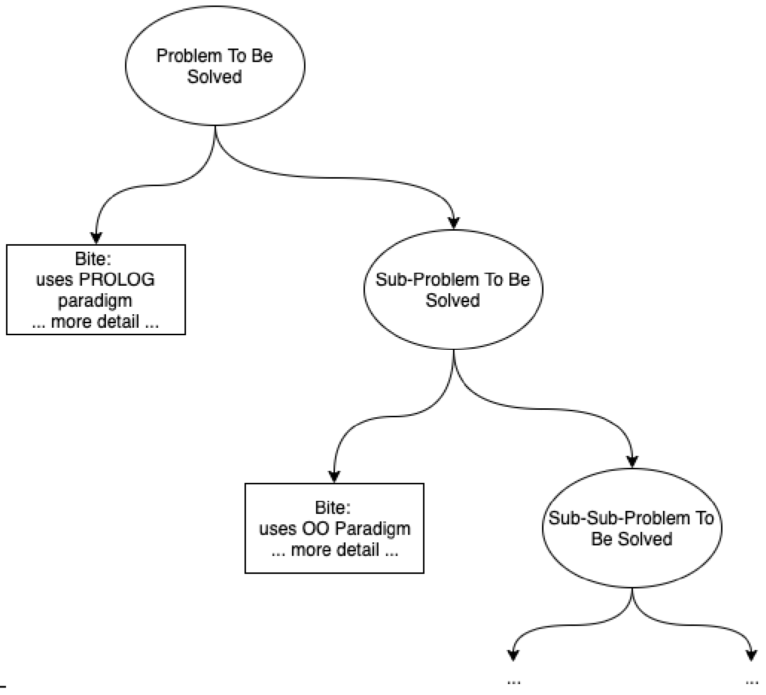
Where does this stop? It depends on the application and the Designer's tolerance for boredom.

As long as the Designer makes the choices clear, expresses the DI[6], then future readers (Optimization Engineers, Maintenance Engineers, Testing Engineers, etc.) can understand – and deal with – the choices made in the design. There is no *one way* to Design something, there is only "here is how I chose to design it" documentation (executable documentation would be more precise than static documentation).

Fractal Design Documentation

Q: Is the documentation for a Design, itself a fractal? A binary tree. Each node contains a bite of the solution and "the rest". Each Bite describes the paradigm an details used for that bite.

For example:


Problem To Be Solved
Bite: uses PROLOG paradigm ... more detail ...
Sub-Problem To Be Solved
Bite: uses OO Paradigm ... more detail ...
Sub-Sub-Problem To Be Solved
...
...

5 N.B. I do not use the phrase CPU, since none of the processors are Central.
6 Design Intent.