2024-04-11 Simplicity - Familiarity

Simplicity

Simplicity is achieved by enabling Flow

Flow is “intuition”.

Flow is the ability to do something without needing to think about it without conscious attention to detail.

Affordances

Affordances are design elements enable application of intuition and bits of flow.

Some flow is better than no flow.

Examples

- Learning to Drive an Automobile

    Learning to drive, at first, seems overwhelming.

    Then, it becomes "second nature" and you can do drive on "autopilot" without thinking about it.

- Learning to Ride a Bicycle

- Learning Intuition About Gravity As A Child

- Putting In Golf

    Geoff Mangum says that all good putters in golf learn to rely on the effects of gravity, for consistency.

If you were transported to the Moon, what would happen to your intuition about tossing a ball?

Where Does Intuition Come From?

Can "intuition" be learned?

Example: my kids think about our refrigerator differently than my parents’ grandparents did.

Example: In the 1980's assembler programmers resisted HLLs. Once assembler programmers retired and died off, programmers began using HLLs exclusively for all programming.

Now, we're waiting for FP'ers to retire and die off, so that we might create new ways to think about software (internet, robotics, gaming, blockchain, GUIs, etc.)

Ptolemaic Cosmology was believed by most of society for 2,000 years. These beliefs were overturned by Galileo only 400 years ago.

What forced us to believe in Ptolemaic Cosmology for such a long period in human history?

Science

Science attempts to explain phenomena in a linear fashion. Everything is considered to be synchronous, even when this does not reflect reality.

Science is
- serialization
- flattening
- anti-layering
- based on "speech", which is a linear form of communication.
- divide and conquer, a methodical breakdown of phenomena.

Programming Using Mathematical Notation

Coding, as currently practiced, is function-based. This includes, but is not restricted to, FP (Functional Programming). Function-based coding includes FORTRAN, Lisp, etc.

Function-based coding appears to have encouraged a conflation between subroutines and mathematical functions.

Subroutines are not functions. Functions, though, can be expressed as a restricted form of subroutines.

Mathematical function notation expresses notions in a manner that is FTL (Faster Than Light).

In collapsing thoughts about a 4D phenomenon (x/y/z/t) down to a 2D notation (x/y), some simplifying assumptions were made. In this case, two dimensions were dropped - z and t (depth and time) in order to allow the notation to be written down on paper. The notation *can* handle the other two dimensions, but, in a second-class manner only. These other dimensions are not built into the notation and must be expressed explicitly,

whereas the two built-in dimensions do not need to expressed explicitly due to the fact that they are built into the notation.

For example, the expressions

```
f(x)
g(y)
```

Are read left-to-right and top-to-bottom - in the *x* direction and in the *y* direction. The concepts of *depth* and *time* are not fully represented in the written form. *Z* is not represented at all - for that we would need a medium that is not a flat sheet of paper, but, is volume-based. *Time* is only partially represented, in two (2) ways:

- The notation specifies that `g(y)` is evaluated only *after* `f(x)` has been evaluated
- The notation specifies that, both, `f(x)` and `g(y)`, *block* waiting for the respective called functions to return values. The actual amount of waiting time is *not* specified. In essence, the written mathematical notation acts in a manner that is faster than light (FTL), whereas an actual implementation of the notation in a computer must obey the laws of physics. On paper, this seems inconsequential, but, in a computer, this makes a difference - and, has caused numerous *gotchas*[1].

Christopher Alexander

Christopher Alexander's book "A Pattern Language ..." Is considered to be influential in the programming community.

There is a huge, unrecognized impedance mismatch between Alexander's underlying assumption and the notation used for programming. Alexander assumes that pattern elements are completely isolated from one another and can be snapped together in a LEGO®-like manner. Programming using function-based notation assumes the opposite, i.e. function-based code requires that all software units be synchronized and strongly-coupled.

Adoption of Alexander's pattern language ideas is hampered simply by the function-based notation used for programming. Implementing truly isolated patterns in code requires extra work and, hence, tends to be avoided, and, ignored, and, mis-implemented, all of which result in confusion and *gotchas* and extra work.

[1] For example, the Mars Pathfinder fiasco was caused by the impedance mismatch between an FTL notation and reality. The fiasco spawned the invention of yet another workaround (aka "epicycle") - that of "priority inheritance".

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*