Background

When I first started riding a motorcycle, I read a book called A Twist of the Wrist. To paraphrase Keith's anology, he says as a rider, you have finite ammount of attention, let's say 10$** , when you first start riding, you spend **3$ to shift gears and **4$** to do a shoulder check **6$** to change lanes. Performing even the most mundane tasks you run out of your budget. As you keep riding the cost performing these tasks go down, like a loyalty scheme, and before you know it, it will cost you **25¢** to shift gears **30¢** cents to do a shoulder check and **80¢** to change lanes, and you more left over with your budget for other tasks, like scanning the traffic and profiling drivers which are inherently more expensive operations.

Thinking back to when I started using **git** I used to search how to revert a commitor how to checkout a new branch, the simplest of taks, what is now a 3 second task, used to take me 4 minutes to perform. The more I used it cheaper it became.

In 1936 Ernst Neufertpublished his seminal work Architects' Data.I came across Architects' Data, or as it's also referred to simply the "Neufert", when I couldn't have been more than 10 years old so roughly around 64 years after the book was published. The cover featured a figure at that age I had mistaken for the Vitruvian Man, which actually made me pick it up and flip through the pages.

Architects' Data is supposed to be used as a reference book when designing buildings, it painstakingly catalogs measurements for buildings categorized by function. The scanned page for instance is shows measurements of an auditorium.

fig 2 shows what happens in an auditorium when a person has to leave a row to get to the aisle. To give you and idea that is just one figure on one page of one category. The book goes has sections on Houses, Flats & Apartments, Gardens, Education, Hospitals, Hotels, Office Buildings, Religious Buildings, Shops and stores, Banks, Vehicle Services, Industrial building, Sports venues, Airports, Farm buildings, Labaratories, Theaters and Cinemas, Museums, Corridors and Stairs, Doors and Windows, Lighting and the list goes on... If we pick just one of those categories, let's take Airports for example, that's further divided in to subcategories like, Planning sequence, Runways/taxiways/terminals, Flow & functions, Baggage handling/aircraft parking and Catering/aircraft maintenance/airfreight

And in every category and it's sub category the attention to detail is at the same level of the above scanned page. As a 10 y/o my imagination went crazy, by than my only other point of reference, remember I'm only 10, was Mythbusters. When Mythbusters folks had to compare a bunch of different things they would go to a shop pick a bunch of different alternatives try them all and write down the results. So as a 10 year old, I started imagining Herr Neufert walking in to auditoriums, hospitals and hotels going "Hallo! wie gehts? Ja, ja! I'm here to measure" when they would ask "what are you going to measure?" he would simply reply "ze everything". Than he'd go randomly sit in chairs, lie down on sofas and/or conveyor betls, stand behind podiums, slide down banisters, punch a whole in a fanta bottle and try out the urinals. I remember thinking, my god that must have been sooooo much work. To this day, I still don't know how Neufert came up with these measurements. I like to hold on to that little piece of my imagination live in ignorance and think that it happened just the way I imagined.

The subject matter of Architects' Data, Ergonomics, plays a big role in it's success and popularity.

The word .I ergonomics was first used by a Polish scientist Wojciech Jastrz bowski in the article titled "Outline of Ergonomics, or the Science of Work Based upon the Truths Drawn from the Science of Nature" published in 1857. It's the combination of the words .B ergo from the Greek word .I érgon meaning work + .B nomics from the Greek word .I nómos meaning rules or law. The first usage of the English word was in July of 1949 by K. F. Hywel Murrell, which led to the establishment of The Chartered Institute of Ergonomics and Human Factors in the September of the same year.

Webster's dictionary defines Ergonomics as;

an applied science concerned with designing and arranging things .B people use so that the .B people and things interact most efficiently and safely. [^ergo]

In desiging ergonomic software if we start with the assumption your users have 10' of attention, what can we do to make them spend less time on a given task?

If they instinctively know how some task might be performed, they are spending less time on them. the less they spend, more likely they are to come back, if they have to spend a hour or a day to perform a simple task, we might price ourselves out of the market because they might not be able to afford our tool even if it's a superior solution.

So over the past couple of years I found myself asking this question over and over again when I'm designing software...

Is this thing I'm building going to make performing a task cheaper?

That is the main motivation behind creating bldy. Can we make an intuitive system that makes performing cetain tasks easier?

Arthitects' Data defines primitives for Architects to tweak and compose in a way that would make it their own. A common vocabulary if you will that all architects use. And this idea is not uncommon, in the books .I A Pattern Language and .I The Timeless Way Christopher Alexander, et al. introduce "the Timeless Way" pattern language.

The elements of this language are entitties called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this sollution a million times over, without ever doing the same thing over and over again.

What is interesting to me is the emergence of such patterns in UI design. Things like Google's Material Designand Apple's Human Interface Guidelinesgive UI designers such measurements and primitives to tweak and compose to build UI systems in a consistent way. We call these Design Systems. Google's

Material, Github's Primer, Zeit's Geist (cause why not I guess)Mozilla's Protocolall converge on things like Cards, Labels, Avatars and Accordions even though these are not standard DOM elements, it is how we've gotten used to interacting with web pages. For the most part we accept these are meaningful solutions to problems that we implicitly trust that exist.

Let's take .I Card component for example. Like I've said .I Card is not a standard DOM element, but in most design languages we see an implementation of it.

I believe that this is one area that design languages fall short, they present solutions without defining the .B problem these component solve?

In his talk from CPPCon 2014 titled "Data-Oriented Design and C++" Mike Acton says, if the problem changes so must the solution. Understanding the problem domain helps us define the problem and come up with an appropriate solution for that particular domain. Future proofing for potential problems that one doesn't have, will result in problems that one will definitely have.

For most, it might be obvious what the Card component does, but making it available as a component might end up with it being misused without the appropriate context of what actual problem it solves.

If were to say

Card pattern is useful for presenting a collection of items of the similar type in an easily distinguishable way.

We would have a clear statement of what problem this pattern is actually trying to solve and it would be less likely to be misused. The fit and finish would be up to the implementor to.

By combining multiple patterns, not solutions, we can define a common vocabulary to communicate larger ideas.

The idea of repeating patterns not solutions in software engineering is not a new idea. It goes back as eaerly as the Unix Philosophy. It is often shortened to "Write programs that do one thing and one thing well" but the Unix Phioshopy[^up] is as follows:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

The problem with reusing code as opposed to patterns in the context of a build systems is that it hides away a bunch of platform specific configurations without it making the code more maintainable. For instance, if we were to create a standard C++ rule for our build system, we would have to take in to account all the different ways in which compiler flags can change the way something is compiled. If we were to take clang as an example, it supports AARCH64, AMDGPU, ARM, Hexagon, MIPS, PowerPC, WebAssembly, X86, RISCV each architechture flags that are specific to that hardware platform. To incorporate all of a platforms concerns in to a single rule-set is extremely hard, documenting all the flags and what they do, if history is any indicator with developers documenting stuff, would simply won't happen.

But there's very little benefit to strongly typing command line flags if you can dynamically generate them based on the platform you are building for.

If you've ever google'd "MapReduce explained", possibly when you were studying for a technical interview, you may have come across something that looks like this.

I like this image a lot, it really cuts down to the core of what map and reduce are.

What's funny is that as developers we use map reduce all the time, and possibly a better example of it is hiding in plain sight.

What won't come as a shock is that every compilation is Map{Filter}Reduce jobs.

We take source files, we compile them in to object files and we reduce them to binaries. And if I were to replace the sandwich breads and tomatoes with source files, and sliced onions with object files we get roughly the same image.

For the most part building software is source code transformation by mapping source files in to some intermediary format and reducing them to a binary, library, docker image or what have you. Sometimes the work we do in the map phase is very light or doesn't exist, and the reduce phase is quite heavy but for the most part any compilation job that can be distributed and paralellized can be described as MapReduce jobs.

For example if we were to compile **hello.c** like so using clang like so

```
clang hello.c -o hello     # compile & link
```

clang would be acting as our build system, compile and link **hello.c** for us. We can verfiy this is the case by passing the **--verbose** flag to the little rascal to see what's hidden under the covers.

```
$ clang --verbose hello.c -o hello
# (ommitted for space some debug info)
 "/usr/bin/clang-8" -cc1 -triple x86_64-pc-linux-gnu -emit-obj -mrelax-all -disable-free -di
# compile       ^^^
 "/usr/bin/ld" -pie --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2
# link  ^^^
```

Sure enough, it is actually running 2 different binaries, one to compile the source and the header file to an intermediary object file, and ld to link it with the default system libraries by passing the **-lc** and **-lgcc** flags.