Effective Programs

- **Speaker: Rich Hickey**
- **Conference: Clojure/Conj 2017 - Oct 2017**
- **Video: https://www.youtube.com/watch?v=2V1FtfBDsLU**

[Time 0:00:00]

```
slide title: Effective Programs

      10 Years of Clojure

          Rich Hickey

         [Clojure logo]
```

I feel like a broken record every time I start these talks by thanking everybody. So I want to start this talk in a different way by saying, "My son is getting married today."

[Audience applause]

"In another state."

[Audience laughter]

So right after I give this talk, I am going to hop on a plane, go do that. I will be back tomorrow morning. So I have not disappeared. I am looking forward to the follow-up talks and everything else, but I will be missing in action briefly.

So now, to be redundant. Thanks everybody for coming. Ten years ago, Clojure was released, and there is no possible way

[Audience applause]

I could have imagined this. I told my wife Steph, "If 100 people use this, that would be ridiculously outrageous". And that is not what happened. And what did happen is interesting. I do not think it is fully understood.

But I wanted today to talk about a look back a little bit about the motivations behind Clojure. It is not like when you come out with a programming language you can tell that whole story. I think one: because it is not good marketing, and two: because if you really want to be honest, you probably do not know. It takes time to understand what happened, and why, and what you really were thinking. And I will not pretend I had a grand plan that incorporated everything that ended up becoming Clojure. It certainly involved a lot of interaction with people in the community.

But, there is this. "Clojure is opinionated". We hear this. And I think it is interesting to think about two aspects of that. One is, in which ways is it? And what does it mean for a language to be opinionated? I think in Clojure's case, people come to it, and they are like "Wow. This is forcing me, everywhere I turn, to do something a certain way".

And I think the nice way to say that is: there are only a few strongly supported idioms, and a lot of support for them. So if you use the stuff that comes with it, there is a whole story that supports your efforts. And if you want to fight against that, we do not do too much.

[Alex was asking me which glasses were the right ones and "neither" is the answer.]

But design is about making choices.

[Time 0:02:53]

```
slide title: Clojure is 'opinionated'

+ How so?
  + Few, strongly supported idioms
  + Choices made
+ Why?
  + Pain of experience
```

And there are a bunch of choices in Clojure. In particular, there is a big choice about what to leave out. And part of this talk will be talking about what was left out.

The other side of being opinionated is, "How do you get opinionated?" I mean it is not like I am opinionated.

[Audience laughter]

Of course I am opinionated, and that comes from experience. When I started

[Time 0:03:16]

```
slide title: Application Development

+ scheduling systems (C++)
+ broadcast automation (C++, Java)
+ audio fingerprinting and recognition (C++)
+ yield management (Common Lisp producing SQL)
+ more scheduling (Common Lisp -> C++)
```

doing Clojure in 2005, I had already been programming for 18 years. So I had had it. I was done. I was tired of it. But I had done some really interesting things with the languages that professional programmers used at the time.

So primarily I was working on scheduling systems in C++. These are scheduling systems for broadcasters. So radio stations use scheduling systems to determine what music they play. And it is quite sophisticated the way that works. You think about "well over the course of the day you do not want to repeat the same song". You actually have to think about the people who listen to the radio for one hour in the morning and this other hour in the afternoon. And you create sort of an alternate time dimension for every drive time hour. Things like that. So there is multi-dimensional scheduling and we used evolutionary program optimization to do schedule optimization.

Broadcast automation is about playing audio. And at the time we were doing this, playing audio on computers was a hard thing. It required dedicated cards to do the DSP work.

I did work on audio fingerprinting. So we made systems that sat in closets and listened to the radio, and wrote down what they heard. And this was both used to track a station's playlist, and then eventually to track advertising, which was where the money was for that. Which involved figuring out how to effectively fingerprint audio and scrub audio, sort of compare novelty to the past.

[Time 0:04:59]

I worked on yield management systems. Does everybody know what "yield management" is? Probably not. So what do hotels, airlines, and radio stations have in common? Their inventory disappears as time passes. "Oh, I have a free room. I have got a slot in my schedule. I have got a seat on this airplane." And then time passes and nobody bought it, and now you do not.

So yield management is the science and practice of trying to figure out how to optimize the value of your inventory as it disappears out from under you. And that is about looking at the past and past

sales, and it is not simplistic. So for instance, it is not an objective to sell all of your inventory. The objective is to maximize the amount of revenue you get from it, which means not selling all of it in most cases.

That was not written in C++. That was around the time I discovered Common Lisp, which was about 8 years into that 15 years. And there was no way the consumer of this would use Common Lisp, so I wrote a Common Lisp program that wrote all the yield management algorithms again out as SQL stored procedures and gave them this database, which was a program.

Eventually I got back to scheduling, and again wrote a new kind of scheduling system in Common Lisp, which again they did not want to run in production. And then I rewrote it in C++. Now at this point I was an expert C++ user and really loved C++, for some value of love

[Audience laughter]

that involves no satisfaction at all.

[Audience laughter]

But as we will see later I love the puzzle of C++. So I had to rewrite it in C++, and it took four times as long to rewrite it as it took to write it in the first place. It yielded five times as much code, and it was no faster. And that is when I knew I was doing it wrong.

[Time 0:07:09]

```
slide title: Application Development (2)

+ US national exit poll, election projection system (C#)
+ Clojure (Java, Clojure)
+ machine listening, artificial cochlea (Common Lisp,
  Mathematica, C++, Clojure)
+ Datomic database (Clojure)
```

I went on to help my friend Eric write the new version of the National Exit Poll System for the U.S., which also involves an election projection system. We did that in a sort of self-imposed functional style of C#.

And then around 2005, I started doing Clojure and this machine listening project at the same time. And I had given myself a 2-year sabbatical to work on these things, not knowing which one would go where. And leaving myself free to do whatever I thought was right. So I had zero commercial objectives, zero acceptance metrics. I was trying to please myself for two years. I just sort of bought myself a break.

But along the way during that period of time, I realized I would only have time to finish one. And I knew how to finish Clojure, and machine listening is a research topic. I did not know if I was two years away or five years away. So Clojure is written in Java and eventually the library is written in Clojure.

And the machine listening work involved building an artificial cochlea, and I did that in a combination of Common Lisp and Mathematica and C++. And in recent years as I have dusted it off, I have been able to do it in Clojure, and that is sort of the most exciting thing. I needed these three languages before to do this and now I only need Clojure to do it.

And then I did Datomic, which was also Clojure.

[Time 0:08:39]

```
slide title: Databases!

+ ISAM
+ SQL
+ RDF
+ LWW (?)
```

Almost all of these projects involved a database. All different kinds of databases, from ISAM databases, a lot of SQL, many attempts but not many integrations of RDF. Databases are an essential part of solving these kinds of problems. It is just what we do.

How many people use a database in what they do every day? How many people do not? OK.

So this last thing [LWW] is not an acronym for a database. It is there to remind me to tell this anecdote.

So I used to go to the Light-Weight Languages workshop. It was a one-day workshop held at MIT where people working on small languages, either proprietary or just domain-specific, DARPA or whatever, would talk about their little languages, and what they were doing with little languages. It was very cool and very exciting. Got a bunch of language geeks in the same room, and there was pizza afterwards.

So I remember, I would just go by myself or with my friend. I was not part of the community that did that. They just let me in. But afterwards, they had pizza. So I sat down at pizza with two people I did not know, and I still do not know their names. And it is good that I do not, because I am going to now disparage them.

[Audience laughter]

They were both computer language researchers. And they were talking, also disparagingly, about their associate, who had somehow fallen in with databases and lost the true way. And one of them sort of sneeringly went to the other and said, "Aw, David, when was the last time you used a database?". And he was like, "I do not know that I have ever used a database."

And like I sort of choked on my pizza, because theoretically they are designing programming languages. And yet they are programming, and they never use databases. I did not know how that worked.

But it was part of the inspiration to do

[Time 0:10:38]

```
slide title: 'Situated' Programs

  situate - "to put in or on a particular site or place"

+ execute for extended periods of time - often continuously
+ deal with information
+ have time-extensive 'memory'
  + remember/recall from databases
+ deal with real-world irregularity
```

Clojure because, if people who do not do databases can write programming languages, *anybody* can.

[Audience laughter]

So there are different kinds of programs. And one of the things I tried to capture on this slide is to talk about what those kinds of programs were that I was working on. And the word I came up with

were "situated" programs. In other words, you can distinguish these kinds of programs that sit in the world and are sort of entangled with the world. They have a bunch of characteristics.

One is: they execute for an extended period of time. It is not just like calculate this result and spit it over there. It is not like a lambda function at AWS. These things run on an ongoing basis, and they are sort of wired up to the world. And most of these systems run continuously, 24/7. It is quite terrifying to me that now these things, which are 30 years old, are almost definitely still running 24/7 somewhere, if they have not been replaced. So this first notion of extended periods of time means continuously, as opposed to just for a burst.

They almost always deal with information. What were the kinds of things that I talked about? Scheduling. In scheduling you look at what you have done in the past. You look at your research data. What does your audience tell you they like, or they are interested in, or what they are burnt out on? And you combine that knowledge to make a schedule.

Yield management looks at the past sales, and sales related to particular periods of time, and facts about that, and produces pricing information.

The election system looks at prior vote records. How did people vote before? That is a big indicator of how they are going to vote again. Of course the algorithms behind that are much more sophisticated.

But in a simplified way, you can say all of these systems consumed information. And it was vital to them. And some of them produced information. They tracked the record of what they did.

[Time 0:12:35]

And that is this next point, which is that most of these systems have some sort of time-extensive memory. That database is not like an input to the system that is fixed. It is something that gets added to as the system runs. So these systems are remembering what they did. And they are doing it both for their own consumption, and for consumption by other programs quite often.

And they deal with real-world irregularity. This is the other thing I think that is super-critical in this situated programming world. It is never as elegant as you think, the real world.

And I talked about that scheduling problem of: there is linear time, somebody who listens all day, and there is somebody who listens just while they are driving in the morning and the afternoon. Eight hours apart there is one set of people, and then an hour later there is another set of people, and another set. You have to think about all that time.

You come up with this elegant notion of multi-dimensional time, and be like, "Oh, I am totally good ... except on Tuesday". Why? Well, in the U.S., on certain kinds of genres of radio, there is a thing called "two for Tuesday". So you built this scheduling system, and the main purpose of the system is to never play the same song twice in a row, or even pretty near when you played it last. And not even play the same artist near when you played the artist, or else somebody's going to say, "All you do is play Elton John. I hate this station".

But on Tuesday, it is a gimmick. "Two for Tuesday" means, every spot where we play a song, we are going to play two songs by that artist, violating every precious elegant rule you put in the system. And I have never had a real-world system that did not have these kinds of irregularities. And where they were not important.

[Time 0:14:28]

```
slide title: 'Situated' Programs (2)

+ interact with other systems
+ often interact with humans
+ remain in use for long periods of time
```

```
+ are situated in a changing world
+ use other people's code
+ _not compilers_
```

Other aspects of situated programs. They rarely are sort of their own little universe where they get to decide how things are, and they do not need to interact with anyone else or agree with anyone else. Almost all these systems interacted with other systems.

Almost all of these systems interacted with people. Somebody would sit there and say, "start playing this song right now", or "skip this song", and we are like "I scheduled that song, and I balanced everything around you playing it, and now your DJ just said 'do not do that' ".

The election projection system has *tons* of screens for users to look at things, and cross-tabulate things, and make decisions, feeding all the things you see on TV, so people can explain things to other people. So people, and talking to people, is an important part of these programs.

They remain in use for long periods of time. These are not throw-away programs, like I said. I do not know that much of the software I ever wrote has stopped being run by somebody. People are still using it.

And they are also situated in a world that changes. So again, your best laid plans are there the day you first write it, but then the rules change. May be there is "three for Thursdays". I do not know, but when that happens, go change everything to deal with it.

Another aspect of being situated, and it is one I have been thinking about a lot more recently is, being situated in the software environment and community. Your program is rarely written from scratch, with all code that you wrote just for the purpose of the program. Invariably, you are going to pull in some libraries. And when you do, you have situated yourself in that library ecosystem. And that is another thing.

So when I talk about situated programs, and you look at the programs I talked about having written in my career, one of them really sticks out, right? What is that?

[Time 0:16:21]

Clojure. Compilers, they are not like this. They do not have a fraction of these problems. They take some input right off the disk. They get to define the whole world, right? When you write a language, what do you do? The first thing you do when you write a language, you get rid of any "two for Tuesdays".

[Audience laughter]

Right? You can just disallow it. You try to make the most regular thing. And then your programming is just, well, now I have to enforce the rules that I made up for myself. It is like, wow, what could be easier than that?

[Audience laughter]

And it really is a lot simpler. They do not generally use a database. Although I think they probably should. They rarely talk over wires and blah, blah, blah, blah, blah. So compilers and theorem provers and things like that are not like these programs.

[Time 0:17:08]

```
slide title: Effective

    producing the intended or expected result

from Latin effectus "accomplishment, performance"
```

So the title of this talk is "Effective Programs". And what does "effective" mean? It means "producing the intended result". And I *really* want this word to become important, because I am really tired of the word "correctness", where "correct" just means, I do not know, "made the type checker happy".

[Audience laughter]

None of my consumers of these programs that I did professionally care about that. They care that the program works, for their definition of "works".

On the other hand, I do not want this to be taken as, "this is a recipe for hacking", just like "do anything that kind of works". So we have to talk about what "works" means. What does it mean to actually accomplish the job of being effective.

[Time 0:17:51]

```
slide title: What is Programming About?

+ making computers effective in the world
+ how are we effective in the world?
  + leverage experience to generate predictive power
  + enabling good decisions and successful activities
+ experience == information == facts about things that actually happened
```

And that is where I want to sort of reclaim the name "programming". Or at least make sure we have a broad definition that incorporates languages like Clojure and the approaches that it takes. Because I think these problems matter.

So what is programming about? I am going to say, for me, "programming is about making computers effective in the world". And I mean "effective" in the same way we would talk about *people* being effective in the world. Either the programs themselves are effective, or they are helping people be effective.

Now, how are we effective? Well, sometimes we are effective because we calculate really well. Like maybe when we are trying to compute trajectories for missiles or something like that.

But mostly not. Mostly areas of human endeavor. We are effective because we have learned from our experience, and we can turn that experience into predictive power. Whether that is knowing not to step in a giant hole or off a cliff, or walk towards the roaring lion, or how to market to people, or what is the right approach to doing this surgery, or what is the right diagnosis for this problem.

People are effective because they learn, and they learn from experience and they leverage that. And so, I am going to say, "being effective is mostly not about computation, but it is about generating predictive power from information".

And you have heard me talk about information. It is about facts. It is about things that happened. Experience, especially when we start pulling this into the programming world, experience equals information, equals facts about things that actually happened. That is the raw material of success. In the world, it is for people. It should be for programs that either support people, or replace people, so they can do more interesting things.

[Time 0:19:45]

```
slide title: What is Programming _Not_ About?

+ itself - that's mathematics
```

```
mathematics may be defined as the subject in which
   we never know what we are talking about, nor
        whether what we are saying is true

-- Bertrand Russell


+ just algorithms/computation
```

So I will also say that, for me, "what is programming not about?" It is not about itself. Programming is not about proving theories about types being consistent with your initial propositions. It is not. That is an interesting endeavor of its own. But it is not what I have been talking about. It is not the things I have done in my career. It is not what programming is for me, and it is not why I love programming. I like to accomplish things in the world.

Bertrand Russell has a nice snarky comment about that. He is actually not being snarky. He wants to elevate mathematics and say, "it is quite important that mathematics be only about itself". If you start crossing the line, and standing on stage and saying, "mathematical safety, type safety equals heart machine safety", you are doing mathematics wrong, according to Bertrand Russell.

And it is not just algorithms and computation. They are important, but they are a subset of what we do.

[Time 0:20:46]

```
slide:

[ rectangle containing the word "Logic" ]
```

So, do not get me wrong. I like logic. I have written those scheduling systems. I have written those yield management algorithms. I have written a Datalog engine. I like logic. I like writing that part of the system. I usually get to work on that part of the system. That is really cool.

[Time 0:21:03]

```
slide:

[ Now there is a circle around the square, just barely surrounding the
square. ]
```

But even a theorem prover, or a compiler, eventually needs to read something from the disk, or spit something back out, print something. So there is some shim of something other than the logic. But in this world of situated programs, and the kinds of programming that I have done, and I think that Clojure programmers do,

[Time 0:21:25]

```
slide:

[ Now the square is smaller, occupying only a small fraction of the
area of the circle.  The square is no longer labeled "Logic", because
the text would be unreadably small.  The circle contains the text
"Information Processing". ]
```

that is a small part of the program. Programs are *dominated* by information processing. Unless they have UIs, in which case, there is this giant circle around this, where this looks like a dot.

[Audience laughter]

But I am not going to go there.

[Audience laughter]

Actually, because I do not do that part. But the information processing actually dominates programs both in the effort ... The irregularity is often there. It is this information part that takes all the irregularity out of the way, so my Datalog engine can have an easy day, because everything is now perfect, because I see a perfect thing, because somebody fixed it before it got to me.

And I do not want to make light of this. I think this is super-critical. Your best [TBD] Google's coolest search algorithm, if they could not get it to appear on a web page and do something sensible when you type something and pressed enter, no one would care. This is where the value proposition of algorithms gets delivered. It is super important. But in my experience,

[Time 0:22:32]

```
slide:

[ Square moved to bottom right of enclosing "Information Processing"
circle, and became a smaller fraction of circle's area then previous
slide. ]
```

while this is the ratio [jump back to previous slide] it probably needs to be to solve the problem, this is the ratio [jump forward to slide with circle being a larger fraction of everything] it often is, and was in my experience in my work.

[His point: Not only was the information processing part the largest part of the program, it was larger than it should have been.]

Actually, this is also sort of bigger. The square would be more of a dot. That the information part of our programs is *much* larger than it needs to be, because the programming languages we had then, and still have mostly, are *terrible* at this. And we end up having to write a whole ton of code to do this job. Because it is just not something the designers of those languages took on.

[Time 0:23:05]

```
slide:

[ Now add several ovals labeled "Libraries" to bottom left of
"Information Processing" circle. ]
```

And of course we are not done. We do not write programs from scratch, so we have to start dealing with libraries. When we do that, now we have started to cross out of "we get to define everything" land. Now we have

[Time 0:23:17]

```
slide:

[ Same as previous picture, but with dotted lines indicating
dependencies from Information Processing code to Libraries, and from
the "Logic" square to to Libraries. ]
```

relationships. And we have to define how we are going to talk to libraries, and how they may talk to us, but mostly we talk to them. So now there are lines. There is some protocol of how do you talk to this library. And we are still not

[Time 0:23:30]

```
slide:

[ Now add a line with arrows in both directions, between the
"Information Processing" circle to/from a box labeled "Database". ]
```

done. Because we said these situated programs, they involve databases. Now, while the information processing and the logic and the libraries may have all shared a programming language – or at least on the JVM, something like the JVM, a runtime – now we are out of that. Now we have a database that is clearly over there.

It is written in a different language. It is not colocated in memory, so there is a wire. It has its own view of the world, and there is some protocol for talking to it.

And invariably, whatever that protocol is, we want to fix it.

[Audience laughter]

And why is that? Well it is something I am going to talk about later called "parochialism". We have adopted a view of the world our programming language put upon us, and it is a misfit for the way the database is thinking about things. And rather than say, "I wonder if we are wrong on our end?" we are like, "Oh no, we have got to fix that. That relational algebra, it cannot possibly be a good idea".

[Time 0:24:36]

OK, but we are still not done. I said these programs, they do not sit by themselves. They talk to other programs. So now, we have three or more of these things. And now they may not be written in the same programming language. They all have their view of the world. They all have their idea of how the logic should work. They all have their idea of how they want to talk to libraries or use libraries. And there are more wires and more protocols.

And *here* we do not get the database vendor at least giving us some wire protocol to start with, that we will fix with ORM. We have to make up our own protocols. And so we do that. And what do we end up with? JSON, right? It is not good.

But at least now

[Time 0:25:20]

we have something . . . So when I program, this is what I am programming. This is a program, to me. This is going to solve a problem, and no subset of this is going to solve the problem. This is the first point you start solving the problem. But you are not done with problems . . .

[Time 0:25:38]

Because it is not a one-shot, one-time, one-moment, one great idea, push the button, ship it, move on, kind of world, is it? Every single aspect of this mutates over time. The rules change. The requirements change. The networks change. The computing power changes. The libraries that you are consuming change. Hopefully the protocols do not change, but sometimes they do. So we have to deal with this over time. And for me, effective programming is about doing this, over time, well.

[Time 0:26:18]

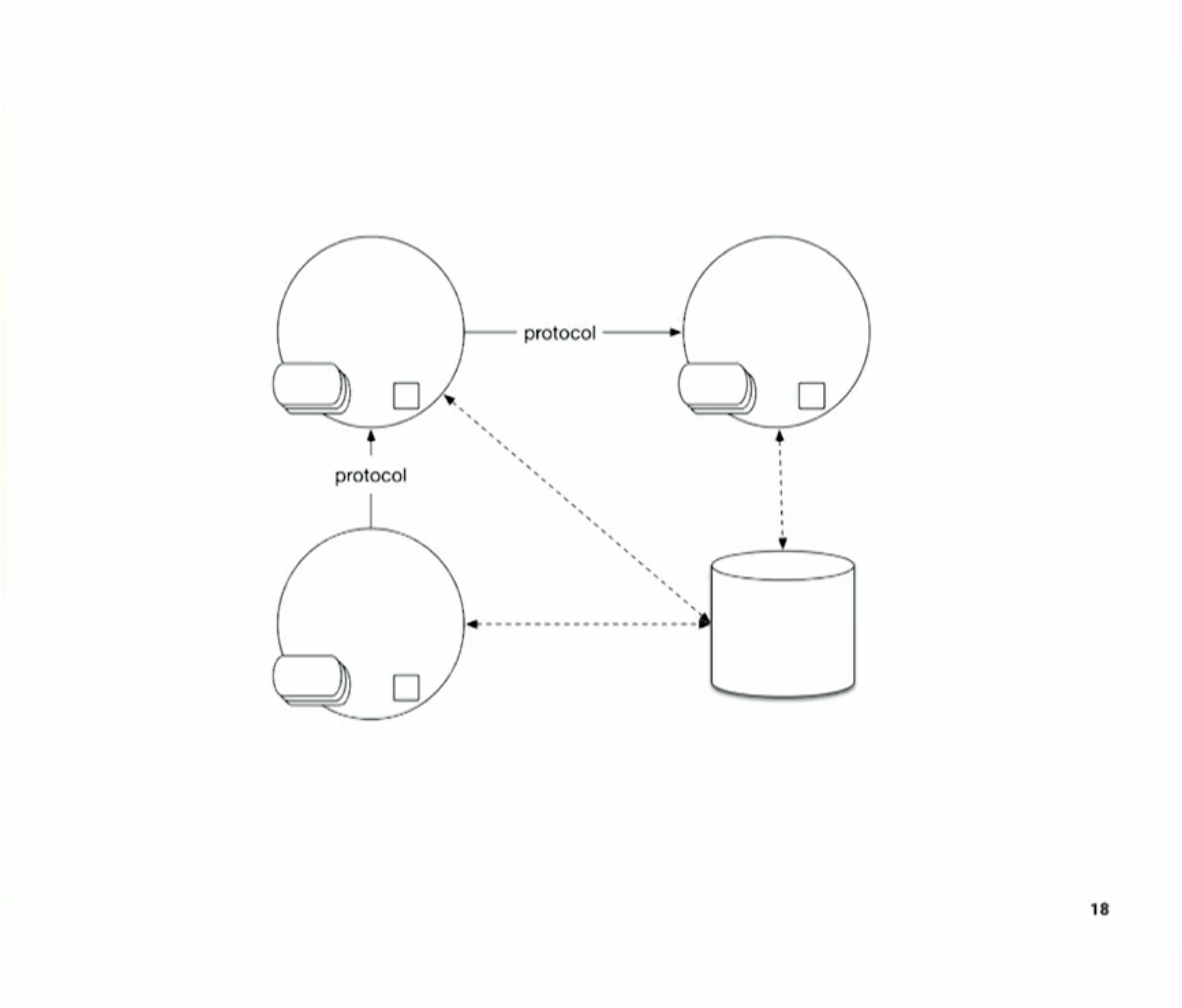

Figure 1: 00.24.36 Three or More...

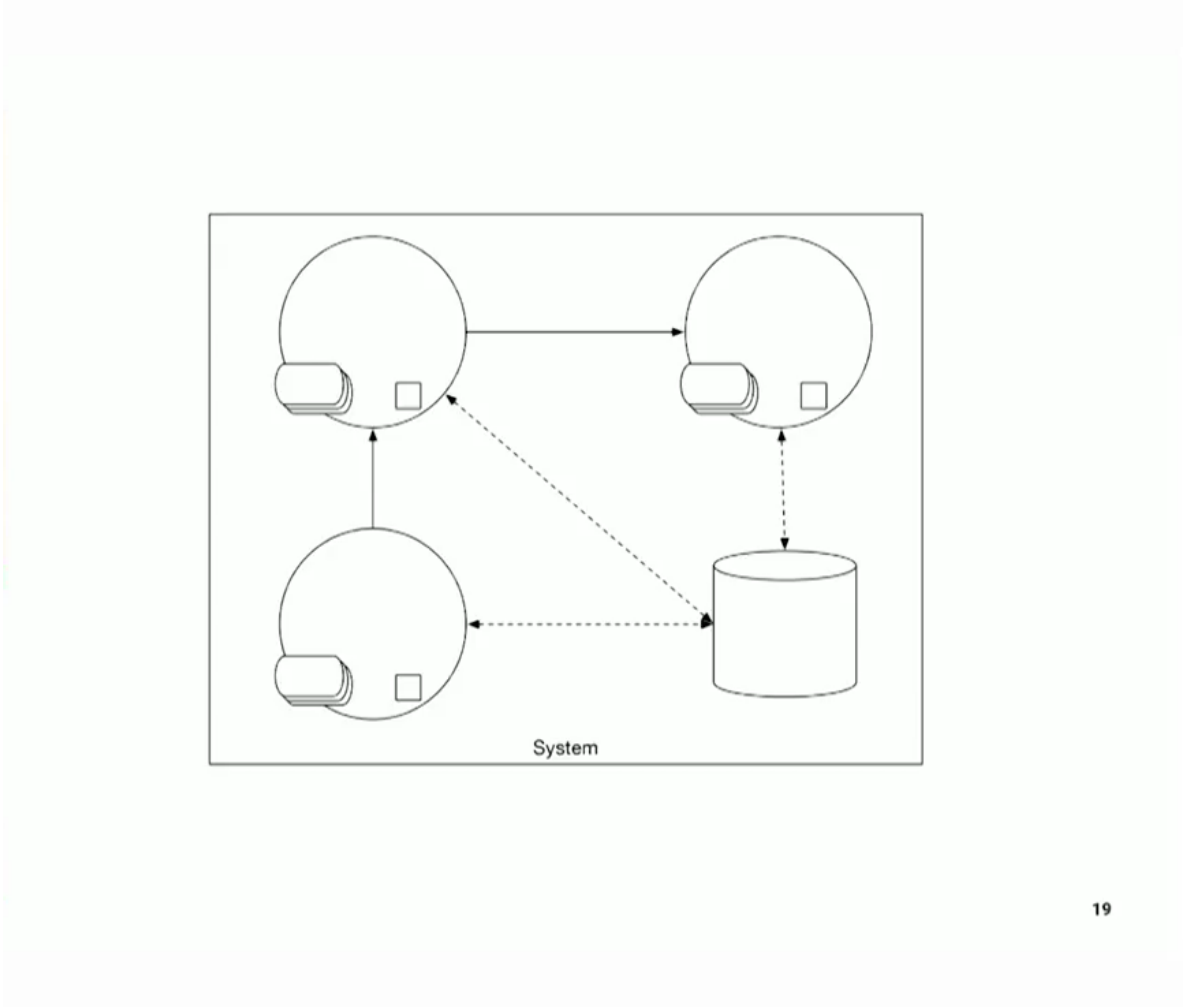

Figure 2: 00.25.20 System

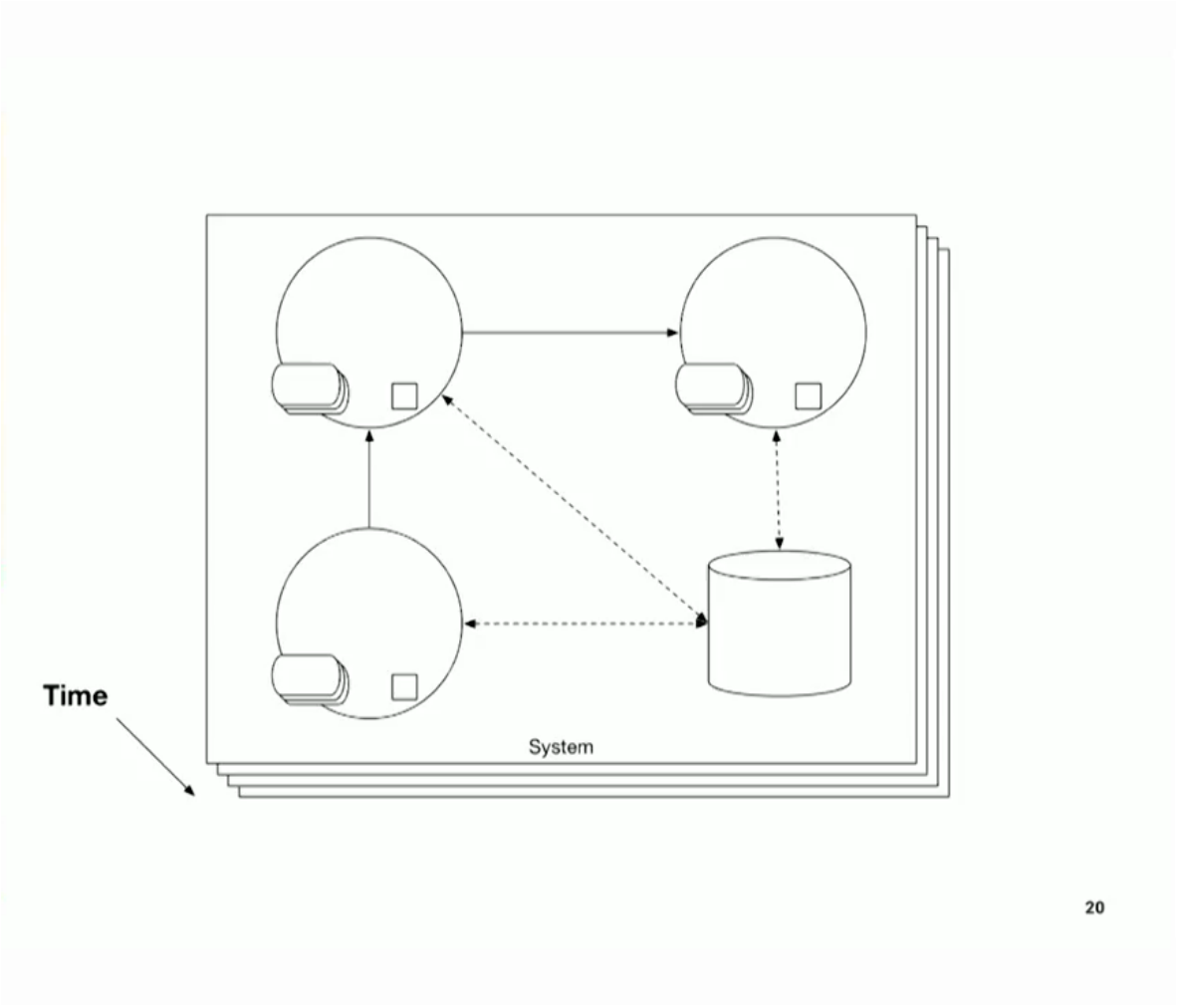

Figure 3: 00.25.38 System + Time

```
slide title: Different Strokes

You'll get a different language if you're writing:

+ compilers, theorem provers
+ device drivers
+ phone switches
+ information-driven situated programs

What are _you_ doing?
```

So I am not trying to say, "there is a right and wrong way, and Clojure is right and everything else is wrong". But it *should* be apparent, and maybe it is not, because I think we all aspire to write programming languages that are general purpose. You could probably write a theorem prover in Clojure. Actually, I am sure you could.

But you certainly would get a different language if your target were compilers and theorem provers, or your target were device drivers, or phone switches. Clojure's target is information-driven situated programs. There is not a catchy phrase for that. But that is what I was doing. All my friends were doing that. How many people in this room are doing that? Yeah. So when you look at programming languages, you really should look at: what are they for? There is no inherent goodness. There are suitability constraints.

[Time 0:27:19]

```
slide title: The Problems of Programming

Domain Complexity
Misconception

10x

PLOP - place oriented programming
Weak support for Information
Brittleness/coupling
Language model complexity
Parochialism/context
weak support for Names
Distribution
Resource utilization
Runtime intangibility
Libraries
Concurrency

10x

Inconsistency
Typos
```

So before I started Clojure, I drew this diagram ... Which I did not. That would have been an amazing feat of prescience. But as I try to pick apart what was Clojure about – because I think there is no reason to write a new programming language unless you are going to try to take on some

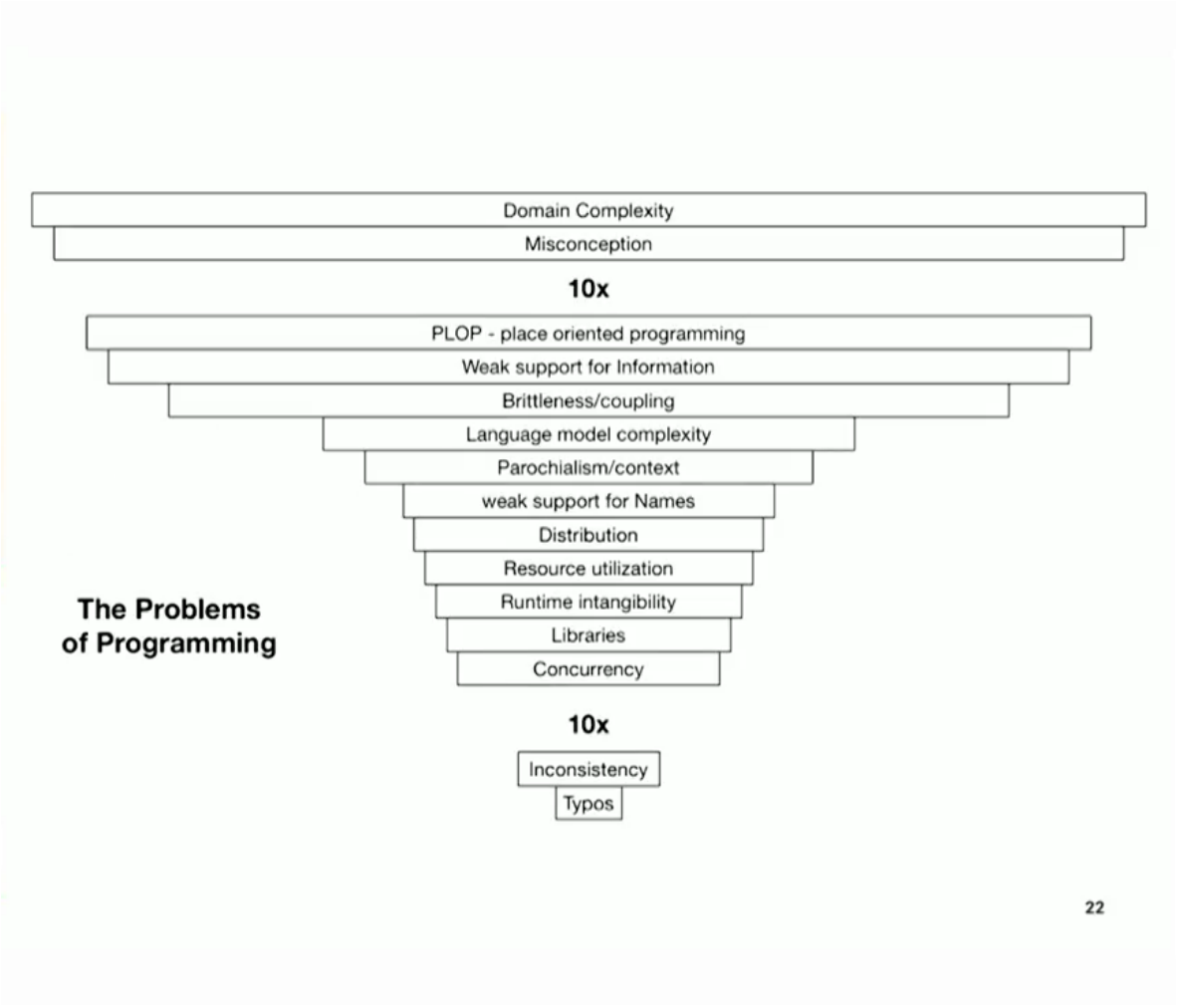

Figure 4: 00.27.19 The Problems of Programming

problems. You should look at what the problems are. I mean, why was I unhappy as a programmer after 18 years and said, "if I cannot switch to something like Common Lisp, I am going to switch careers". Why am I saying that? I am saying it because I am frustrated with a bunch of limitations in what I was using.

And you can call them problems, and I am going to call them the problems of programming. I have ordered them here in terms of severity. And severity manifests itself in a couple of ways. Most important, cost. What is the cost of getting this wrong? At the very top you have the domain complexity, about which you could do nothing. This is just the world. It is as complex as it is.

But the very next level is where we start programming. We look at the world and say, "I have got an idea about how this is, and how it is supposed to be, and how my program can be effective about addressing it". And the problem is, if you do not have a good idea about how the world is, or you cannot map that well to a solution, everything downstream from that is going to fail. There is no surviving this misconception problem. And the cost of dealing with misconceptions is incredibly high.

So then there is this 10x, a full order of magnitude reduction in severity, before we get to this set of problems I think are more in the domain of what programming languages can help with. And because you can read these, they are all going to come up in a second as I go through each one on some slide, so I am not going to read them all out right now.

But importantly, there is another break where we get to *trivialisms* of problems in programming. Like typos and just being inconsistent. Like, you thought you are going to have a list of strings and you put a number in there. That happens. People make those kinds of mistakes. They are pretty inexpensive.

[Time 0:29:49]

[Same as previous slide, except the title has changed, and the lines I have marked with _ are highlighted in green on the slide.]

```
slide title: The Problems Taken on by Clojure

Domain Complexity
Misconception

10x

_PLOP - place oriented programming_
_Weak support for Information_
_Brittleness/coupling_
_Language model complexity_
_Parochialism/context_
_weak support for Names_
_Distribution_
Resource utilization
_Runtime intangibility_
Libraries
_Concurrency_

10x

Inconsistency
Typos
```

So what were the problems that Clojure took on? These green ones. And again I will go through all the green ones in a moment, but I would say, amongst the ones in the middle, I do not think that

Clojure tried to do something different about resource utilization than Java did. It sort of adopted that runtime and its cost model.

I wanted Clojure to be a good library language, but I did not think about the library ecosystem problems as part of Clojure. And my talk last year about libraries implies that I still think this is a big problem for programs. It is one of the ones that is left, right? After you do Clojure and Datomic, what is left to fix?

[Audience laughter]

And the libraries are there. But the inconsistency and typos, not so much. I mean we know you can do that in Clojure. It is actually pretty good at letting you make typos.

[Time 0:30:46]

```
slide title: Clojure Design Objectives

+ can we create effective, situated programs out of substantially
  simpler stuff?
+ with low cognitive load from language
+ make a Lisp I can use instead of Java/C#
```

So fundamentally, what is Clojure about? Can we make programs out of simpler stuff? I mean, that is the problem. After 18 years of using C++ and Java, you are exhausted. How many people have been programming for 18 years? OK. How many for more than 20 years? More than 25? Fewer than 5? So that is really interesting to me. It may be an indictment of Clojure as a beginner's language, or it may be that Clojure is the language for cranky, tired, old programmers.

[Audience laughter and applause]

And you know what? I would not be embarrassed if it was. That is fine by me. Because I did make it for myself, which I think is an important thing to do. Trying to solve other people's problems, and think you understand what they are, that is tricky.

So when I discovered Common Lisp, having used C++, I said that, "I am pretty sure the answer to this first question is, 'yeah, absolutely' ". And can we do that with a lower cognitive load? I also think, "yes, absolutely". And then the question is, "can I make a Lisp I can use instead of Java or C#?" Because you just heard my story, and I used Common Lisp a couple of times, and every time it got kicked out of production. Or just *ruled* out of production, really not kicked out. It did not get a chance. So I knew I had to target a runtime that people would accept.

[Time 0:32:23]

```
slide title: Meta problems

+ acceptability
  + performance
  + deployment platform
+ power
  + leverage
  + compatibility
```

So there are these meta problems. You can try to take on some programming problems, but there are always problems in getting a language accepted. I did not think Clojure would get accepted. Really, honestly.

But I knew, if I wanted my friend, who thought I was crazy even doing it, person number one other than myself to try it, I would have to have a credible answer to the acceptability problems and the power problems. Because otherwise it is just not practical. It is like, "That is cool Rich, but, like, we have work to do".

[Audience laughter]

"If we cannot use this professionally, really it is just a hobby". So we have acceptability, I think that goes to performance, and for me, I thought it was also the deployment platform.

There is a power challenge that you have to deal with, and that is about leverage and I will talk about that later. And also compatibility. Again that is part of acceptability. But Clojure's ability to say "it is just a Java library, kind of", was big. How many people snuck Clojure into their organization to start with? Right, OK. Success!

[Audience laughter]

[Time 0:33:33]

```
slide title: Non-problems

+ Lisp syntax (parentheses)
+ Few static type checks
```

And then there were other things I considered to be *absolute* non-problems. And the first of these is the parentheses. How many people ... and it is OK to admit, right? Everybody has a story. How many people thought the parentheses were going to be a problem and now think that was crazy thinking? Yeah. Which is fine, I think everybody goes through that. Everybody looks at Lisp and is like, "this is cool but I am going to fix this part before I get going. Before I start, before I understand the value proposition of it at all, I am going to fix this", and that says something about programmers. I am not sure exactly what.

[Audience laughter]

But I do not believe this is a problem, and in fact when we get to the middle of this talk you will see that I think this is the opposite of a problem. This is the core value proposition of Clojure. And I think things like, par-make-it-go-away, whatever that is, it is a terrible idea. And it is not good for beginners to do that, to try to solve a problem that is a feature.

The other thing I considered not a problem is it being dynamic. I worked in C++. We had a thing where we said in C++ is that, "if it compiles it will probably work". Like they say of Haskell. And it was equally true then as it is now.

[Audience laughter]

But we really did believe it! We totally did. And it does not help. It really does not help for the big problems. The top, the big wide ones.

[Time 0:35:04]

```
slide title: PLOP - Place Oriented Programming

#1 self-inflicted programming problem

+ Make FP the default idiom
+ immutable persistent data structures the default
  + Bagwell's HAMT made persistent, 1-2x read, 2-4x write
```

```
+ large library of pure functions
+ immutable local bindings
```

OK, so problem number one on that list was place oriented programming. Absolutely, this is the problem. Almost all the programs I wrote, lots of the things on that list were multi-threaded programs. They are crazy hard in C++. Just impossible to get right, when you adopt a normal mutability approach, mutable objects. So, this is the number one self-inflicted programming problem. It seemed clear to me that the answer was to make functional programming and immutable data the default idiom.

So the challenge I had was: were there data structures that would be fast enough to say, "we could swap this for that"? And the goal I had was to get within 2x for reads and 4x for writes. And I did a lot of work on this. This was actually the main research work behind Clojure, was about these persistent data structures.

And eventually I found ... I looked at Okasaki's stuff and the *fully* functional approach, and none of that gets here. And then I found Bagwell's structures, which were not persistent, but I realized could be made so. And they just have tremendously great characteristics, combining the persistence with the way they are laid out, the way memory works. They made it. They made this bar, and I was able to get my friend to try my programming language.

And we have this large library of pure functions to support this, and immutable local bindings. Basically if you fall into Clojure, your first hurdle is not the parentheses, right? It is this functional paradigm. Everything is gone. There is no mutable variables. There is no state. There are no mutable collections and everything else. But there is a lot of support. There is a *big* library. You just have to sort of learn the idioms. So I think this was straightforward.

The critical thing that is different about Clojure is, by the time I was doing Clojure, the people who invented this stuff had adopted a lot more. I think most of the adherents in the functional programming community considered functional programming to be about *typed* functional programming. Statically typed functional programming *is* functional programming.

And I do not think so. I think that this falls clearly in the 80/20 rule. And I think the split here is more like 99/1. The value props are all on this side, and I think Clojure users get a sense of that. They get a feel for that. *This* is the thing that makes you sleep at night.

[Time 0:37:47]

```
slide title: Information

+ sparse
+ open
+ incremental/accumulative
+ names capture semantics
+ (should be) composable
```

OK, problem number two – and this is the most subtle problem. This is the thing that annoys me the most about statically typed languages – is they are terrible at information. So let us look at what information is. Inherently, information is sparse. It is what you know. It is what happened in the world. Does the world fill out forms, and fill everything out for you? All the things you would like to know? No! It does not. It does not, and "not ever" is probably more correct.

The other thing is, "what can you know?" What are you allowed to know? There are no good answers to that. Whatever you want, right? It is open. What else is there? What *is* there to know? Well I mean, what time is it, right? Because every second that goes by, there is more stuff to know, more

things happen, more facts, more things happen in the universe. So information accretes, it just keeps accumulating.

What else do we know about information? We do not really have a good way of grappling with it, except by using names. When we deal with information as people, names are super-important. If I just say, "47", there is no communication going on yet. We have to connect it.

And then the other big thing, and this is the thing I struggle with so often. I have a system. I made a class or a type about some piece of data. Then over here, I know a little bit more data than that. Do I make *another* thing that is like that? If I have derivation, do I derive to make that other thing? What if I am now in another context and I know part of one thing and part of another thing. What is the type of part of this and part of that? And then there is this explosion.

But of course these languages are doing this wrong. They do not have composable information constructs.

[Time 0:39:44]

```
slide title: The Information Programming Problem

+ classes and algebraic types are terrible for this
  + names not there, or not first class
  + no compositional algebra
  + aggregates determine semantics
    + not how information works
  + yield information 'concretions'
```

So what is the problem with programming in a way that is compatible with information? It is that we elevate the *containership* of information to become the semantic driver. We say, "this is a person, and a person has a name, and a person has an email, and a person has a social security number", and there is no semantics for those three things except in the context of the person class or type, whatever it is.

And often, depending on the programming language, the names are either not there. If you have got these product types where it is like, "person is string x, string x, int x, string x, string x, int x, float x, float", product type. Like, a complete callous disregard for people, names, human thinking. It is crazy.

Or your programming language maybe has names, but they compile away. They are not first class. You cannot use them as arguments. You cannot use them as look-up vectors. You cannot use them as functions themselves.

There is no compositional algebra in programming languages for information. So we are taking these constructs that I think were there for other purposes. We have to use them because it is all we were given, and it is what is idiomatic. Take out a class, take out a type, and do this thing.

But the most important thing is that the aggregates determine the semantics, which is dead wrong. If you fill out a form, nothing about the information you put on that form is semantically dominated by the form you happen to fill out. It is a collecting device. It is not a semantic device, but it becomes so.

And what happens is you get these giant sets of concretions around information. People that write Java libraries, you look at the Java Framework. It is cool. It is relatively small, and everything's about sort of mechanical things. Java's good at mechanical things – well, mechanisms.

But then you hand the same language to the poor application programmers who are trying to do this information situated program problem, and that is all they have got. And they take out a class for

like everything they need, every piece, every small set of information they have. How many people have ever seen a Java library with over 1500 classes? Yeah, everybody.

And this is my experience. In my experience, it does not matter what language you are using. If you have these types, and you are dealing with information, you are going to have a proliferation of non-composable types, that each are a little parochialism around some tiny piece of data that does not compose. And I am really not happy with this.

In programming literature, the word "abstraction" is used in two ways. One way is just like, "naming something is abstracting". I disagree with that. Abstracting really should be drawing from a set of exemplars some essential thing. Not just naming something. And what I think is actually happening here is we are getting not data abstractions, you are getting data concretions.

Relational algebra, that is a data abstraction. Datalog is a data abstraction. RDF is a data abstraction. Your person class, your product class, those are not data abstractions. They are concretions.

[Time 0:43:24]

```
slide title: Clojure and Information

'just use maps'
+ first class, functional associative data structures
+ maps are functions of keys, keywords/symbols fns of maps
+ generic library (algebra, composition, etc)
+ names are first-class, namespace-qualified
+ associating semantics with attributes, not aggregates
```

So we know in practice, Clojure says, "just use maps". What this meant actually was, "Clojure did not give you anything else", right?

[Audience laughter]

There was nothing else to use. There were no classes. There was not a thing to say deftype. There were not types. There was not algebraic data types, or anything like that.

There were these maps, and there was a huge library of functions to support them. There was syntactic support for it. So working with these associative data structures was tangible, well-supported, functional, high-performance activity.

And they are generic. What do we do in Clojure if we have just some of the information here and just some of the information there, and we need both those things over there? We say, "what is the problem?" There is no problem. I take some information, some information, and I merge them, I hand it along. If I need a subset of that, I take a subset of that. I call "select-keys" and I get a subset. I can combine anything that I like. There is an algebra associated with associative data.

The names are first class. Keywords and symbols are functions. They are functions of associative containers. They know how to look themselves up. And they are reified, so you can tangibly flow them around your program and say, "pick out these three things" without writing a program that knows how to write Java or Haskell pattern matching to find those three things. They are independent of the program language. They are just arguments. They are just pieces of data. But they have this capability.

And the other thing, which I think is a potential of Clojure – it is realized to varying degrees, but the raw materials for doing this are there – is that we can associate the semantics with the attributes, and not with the aggregates. Because we have fully qualified symbols and keywords. And obviously spec is all about that.

[Time 0:45:23]

```
slide title: Brittleness/coupling

+ flowing type information and structure major source of coupling
+ positional semantics, parameterization _don't scale_

Clojure emphasizes
+ dynamic typing, no burden of proof
+ open constructs, runtime polymorphism
+ open maps, need-to-know, accept and propagate more
```

All right, brittleness and coupling. This is another thing that is just my personal experience: that static type systems yield much more heavily coupled systems. And that a big part of that time aspect of the final diagram of what problem we are trying to solve, is dominated by coupling when you are trying to do maintenance. Flowing type information is a major source of coupling in programs. Having pattern matching of a structural representation in a hundred places in your program is coupling. Like, this stuff, I am seizing up when I see that. The sensibilities you get after 20 years of programming, you hate coupling. It is like the worst thing, and you smell it coming and you want no part of it. And this is a big problem.

The other thing I think is more subtle, but I put it here because unless you see this, is positional semantics do not scale. What is an example of positional semantics? Argument lists. Most languages have them, and Clojure has them, too. Who wants to call a function with 17 arguments? Nope.

[Audience laughter]

There is one in every room.

[Audience laughter]

Nobody does. We all know it breaks down. Where does it break down? Five, six, seven? At some point, we are no longer happy. But if that is all you have, if you only have product types, they are going to break down every time you hit that limit.

How many people like going to the doctor's office and filling out the forms. Don't you hate it? You get this big lined sheet of paper that is blank. Then you get this set of rules, that says, "put your social security number on line 42, and your name on line 17". That is how it works, right? That is how the world works. That is how we talk to other people? No! It does not scale. It is not what we do. We always put the labels right next to the stuff and the labels matter. But with positional semantics we are saying, "No, they don't. Just remember the third thing mean this, and the seventh thing means that".

[Time 0:47:29]

And types do not help you. They do not really distinguish this, float x, float x, float x, float x, float ... At a certain point, it is not telling you anything. So they do not scale but they – it occurs in other places, so we have argument lists. We have product types. Where else? Parameterization. Who has seen a generic type with more than 7 type arguments in C++ or Java? Yeah, we tend not to see it in Java because people give up on parameterization.

[Audience laughter]

And what do they switch to? Spring!

[Audience laughter]

No, I mean, that is not a joke. That is just a fact. They switched to a more dynamic system for injection. Because parameterization does not scale. And one of the reasons why it does not scale is:

there are no labels on these parameters. They may get names by convention, but they are not properly named. When you want to reuse the type with parameters, you get to give them names

[audio cut off, only for a few seconds]

again. Just like in pattern matching. That is terrible. That is a terrible idea. And it does not scale.

So anywhere positionality is the only thing you have got, you are eventually going to run out of steam. You are going to run out of the ability to talk to people, or they are going to run out of the ability to understand what you are doing.

[Time 0:48:59]

So I think types are an anti-pattern for program maintenance and for extensibility, because they introduce this coupling and it makes programs harder to maintain, and even harder to understand in the first place.

So Clojure is dynamically typed. You do not have this burden of proof. You do not have to prove that, because I made something here and somebody cares about it over there, every person in the middle did not mess with it. Mostly, they do not mess with it. I do not know what we are protecting against, but we can prove now that they are still strings over there.

The constructs are open. We much prefer runtime polymorphism, either by multi-methods or protocols, to switch statements, pattern matching, and things like that.

The maps are open. They are need-to-know. What do we do in Clojure if we do not know something? We just leave it out. We do not know it. There is no maybe this, maybe that. If you actually parameterized the information system, it would be maybe everything. Maybe everything no longer is meaningful. It just isn't.

And nothing is of type maybe something. If your social security number is a string, it is a string. You either know it or you don't. Jamming those two things together, it makes no sense. It is not the type of the thing. It may be part of your front-door protocol, that you may need it or not. It is not the type of the thing.

So the maps are open. We deal with them on a need-to-know basis and you get into the habit of propagating the rest. Maybe you handed me more stuff. Should I care? No. The UPS truck comes and my TV is on the truck. Do I care what else is on the truck? No. I do not want to know. But it is OK that there is other stuff.

[Time 0:50:54]

```
slide title: Language Model Complexity

+ reserve brains for domain problems, not meta-puzzles
+ Clojure is just (functional core of) Lisp
  + small!
+ execution model akin to Java
```

So the other problem is language model complexity. C++ is a very complex language, and so is Haskell, and so is Java, and so are most of them. Clojure is very small. It is not quite Scheme small, but it is small compared to the others. And it is just the basic lambda calculus kind of thing with immutable functional core. There are functions. There are values. You can call functions on values and get other values. That is it. There is no hierarchy. There is no parameterization. There is no existential types, blah blah blah blah blah.

And the execution model is another tricky thing. We are getting to the point, even in Java, where it gets harder and harder to reason about the performance of our programs, because of resources. And

that is unfortunate. At least one of the nice things about C was: you knew if your program crashed, it was your problem. And you just figure it out. But you knew what it was going to take up in RAM, and you could calculate things, and it was quite tractable.

And that matters to programmers. Programming is not mathematics. In mathematics you can swap any isomorphism for any other. In programming you get fired for doing that.

[Audience laughter]

It is different. Performance matters. It is part of programming. It is a big deal. So making this something at least I could say, "it is like Java, and blame them".

[Audience laughter]

Well it is fine. But it also meant all the tooling helped us. All the Java tooling works for Clojure. How many people use YourKit and profilers like that on Clojure? That is pretty awesome to be able to do that.

[Time 0:52:43]

```
slide title: Parochialism - names

RDF got it right

  RDF has features that facilitate data merging even if the underlying
  schemas differ, and it specifically supports the evolution of schemas
  over time without requiring all the data consumers to be changed.

+ subject/predicate/object
  + semantics associated with the predicates/attributes, named by URIs
```

[One dictionary's definition of "parochialism": a limited or narrow outlook, especially focused on a local area; narrow-mindedness".]

Now we are into the real nitty-gritty of things I did not like, and therefore I left out. This type thing, it goes everywhere. And the name I came up for it is "parochialism". This idea that "I have this language, and it is got this cool idea about how you should think about things. You should think about things using algebraic data types. Or you should think about things using inheritance".

It yields this intense parochialism. You start to have representations of things, manifestations of representations of information that they only make sense in the context of this language's rules for things. And they do not combine with anybody else's ideas. You smash against the database. You smash against the wire. You smash against this other programming language, because you have got this idiosyncratic, local view of how to think about things.

RDF did this right. And they did it because they had this objective. They are trying to accomplish something. We want to be able to merge data from different sources. We do not want the *schemas* to dominate the semantics. How many people have ever gotten the same piece of mail from the same company and been like, "What is wrong with your databases, dudes?" Yeah. What is wrong?

What is wrong is, one company bought another company. Now they are the same company. They now have these two databases. In one database, your name is in the person table, and in another database your name is in the mailing list table. Who knows that mailing list table name and person name are actually the same piece of information? Nobody. They have to have meetings. I mean this is a big dollar – this is a big ticket problem. It is not a small ... It is not a laughing matter, right?

[Audience laughter, after Rich started laughing first]

These big companies have giant jobs trying to merge these systems, because table parochiality, it is the same as classes and algebraic data types. It is the same problem. It is not a different problem. It is all like, I had this view of the world, and on the day I decided how the world is, I decided that names were parts of person, and you decided that names were parts of mailing lists. And now we need to fix this. And you know how a lot of those companies fix it? They introduce a third database, usually an RDF database, as a federation point, so they now can figure out these two things are the same. And eventually they will stop sending you the same piece of mail twice.

So there is a subject, predicate, object, and obviously you can see the influence of this on Datomic.

[Time 0:55:30]

```
slide title: Parochialism - types and contexts

+ The more elaborate the type system, the more parochial the types
+ Encourage tyranny of the container
  + Are SSN, email really 'owned' by some class/type?
+ :keys are similar, context needed for interpretation
+ Thwarts info composition
+ Raises bar for program-manipulating programs
```

But it goes further. I would say that the more elaborate your type system is, the more parochial your types are. The less general they are, the less transportable they are, the less understandable by other systems they are, the less reusable they are, the less flexible they are, the less amenable to putting over wires that they are, the less subject to generic manipulation that they are.

Almost every other language that deals with types encourages this tyranny of the container I talked about before. We have a choice in Clojure, I think people go either way. There are two things, one is the container dominates, the other is just sort of the notion of context dominating the meaning, like, "because I called it this in this context, it means that".

But we have the recipe in Clojure for doing better than that, in which you use namespace-qualified keys. With namespace-qualified keys we now can merge data and know what things mean, regardless of the context in which they are used.

And anything about this thwarts the composition I talked about before.

And in particular, because we are pointed at this program-manipulating-program idea, as you will see later, it makes this harder.

[Time 0:56:41]

```
slide title: Clojure Names

+ keywords and symbols distinct scalars (as in some other Lisps)
+ don't disappear, not compiled away into offsets
+ (optionally) namespace qualified - org.myorg/myname
+ reversed domain name system, independent of per-lang
  modules/namespaces/packages
+ aliases to make it easy to do the right thing
```

So Clojure has names that are first class. This is stuff that was in Lisp. It just dominates more because they became the accessors for the associative data type. They are functions in and of themselves. Keywords being functions is sort of the big deal.

They do not disappear. They are not compiled away into offsets. We can pass them around. We can write them down. A user who does not know Clojure can actually type one into a text file and save it, and do something meaningful with our program without learning Clojure.

We have this namespace qualification. If you follow the conventions – which unfortunately a lot of Clojure libraries are not yet doing – of this reversed domain name system, which is the same as Java's, all Clojure names are conflict-free, not only with other Clojure names, but with Java names. That is a *fantastically* good idea, and it is similar to the idea in RDF of using URIs for names.

And the aliases help make this less burdensome. And we have done some more recently to do more with that.

[Time 0:57:45]

```
slide title: Distribution

+ what goes over wires?
  + very basic data types - maps of scalars/vectors/maps
+ program inside as you do outside
```

Then there is this distribution problem. And here is where I start saying, "Taking a language-specific view of program design is a terrible mistake", because you are in that little box. You are ignoring this big picture. As soon as you step back, now you have this problem. You have to talk over wires. How many people use remote object technology? I am really sorry.

[Audience laughter]

Because it is brutal. It is very brutal. It is incredibly brittle and fragile and complex and error-prone and specific. How many people use that kind of technology to talk to people not under their own employ? No, it does not work. That is not how the Internet works. Distributed objects failed. The Internet is about sending plain data over wires. And almost everything that ever dealt with wires only succeeded when it moved to this.

And this is very successful. Why should we program in a way that is all super-parochial if we only need to eventually represent some subset of a program – maybe a subset we did not know in advance – over wires. If we program this way all the time, we program the inside of our programs as "let us pass around data structures", and then somebody says, "Oh, I wish I could put half of your program across a wire, or replicated over six machines", what do we say in Clojure? That is great. I will start shipping some edn across a socket and we are done. As opposed to, you have got to do everything over.

[Time 0:59:22]

```
slide title: Runtime tangibility

+ Smalltalk
+ Common Lisp
+ the JVM
+ Situated sensibilities
```

So there were plenty of inspirations and examples for me of this. Runtime tangibility is one of the things I really got excited about when I learned Common Lisp coming from C++. Smalltalk and Common Lisp are languages that were obviously written by people who were trying to write programs for people. These are not language theoreticians. You can tell. They were writing GUIs. They were

writing databases. They were writing logic programs, and languages also. But there is a system sensibility that goes through Smalltalk and Common Lisp that is undeniable.

And when you first discover them, especially if you discover them late as I did, it is stunning to see. And I think it is a tradition that has largely been lost in academia. I just do not see the same people making systems *and* languages together. They have sort of split apart. And that is really a shame, because there is so much still left to pilfer from these languages.

They were highly tangible. They had reified environments: all the names you could see, you could go back and find the code. The namespaces were tangible. You could load code at runtime. I mean, one thing after another after another.

And the old Perlis quip about, "Any sufficiently large C or C++ program has a poorly implemented Common Lisp", is so true. Again, Spring, right? As you get a larger system that you want to maintain over time, and deal with all those complexities that I showed before, you want dynamism. You *have* to have it. It is not like an optional thing. It is necessary.

[By "the old Perlis quip" Rich is probably referring to Philip Greenspun's tenth rule of programming:

"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

https://en.wikipedia.org/wiki/Greenspun%27s_tenth_rule

Alan Perlis undeniably has some great quotable quotes about programming, too. One of the most commonly quoted ones, related to Clojure, is number 9 from the list at the link below: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

https://web.archive.org/web/19990117034445/http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html
]

But what was particularly interesting for me in implementing Clojure was how much runtime tangibility and situated sensibilities were in the JVM design. The JVM is actually a *very* dynamic thing. As much as Java looks like say, C# or C++, the JVM, it was written with an idea of "We are going to embed these programs on set top boxes, and network them, and need to send code around that you could update their capabilities". That is like, it is situated everywhere you turn. And the runtime has got a ton of excellent support for that. Which makes it a great platform for languages like Clojure.

And thank goodness that the work that the people did on Self, and it did not die, that it actually got carried through here. Not everything did. But it is quite important. And it will be a sad day when somebody says, "Well let us just replace the JVM with some static compilation technology". And I will tell you, targeting the JVM and the CLR, it is plain: the CLR is static thinking and the JVM is dynamic thinking.

So there is situated sensibilities in all of these.

[Time 1:02:26]

```
slide title: Concurrency

+ Functional + immutable gets you 90% there
```

The last problem on my initial slide was concurrency, and I think mostly concurrency gets solved by being functional and immutable by default.

[Time 1:02:36]

```
slide title: Epochal time model
```

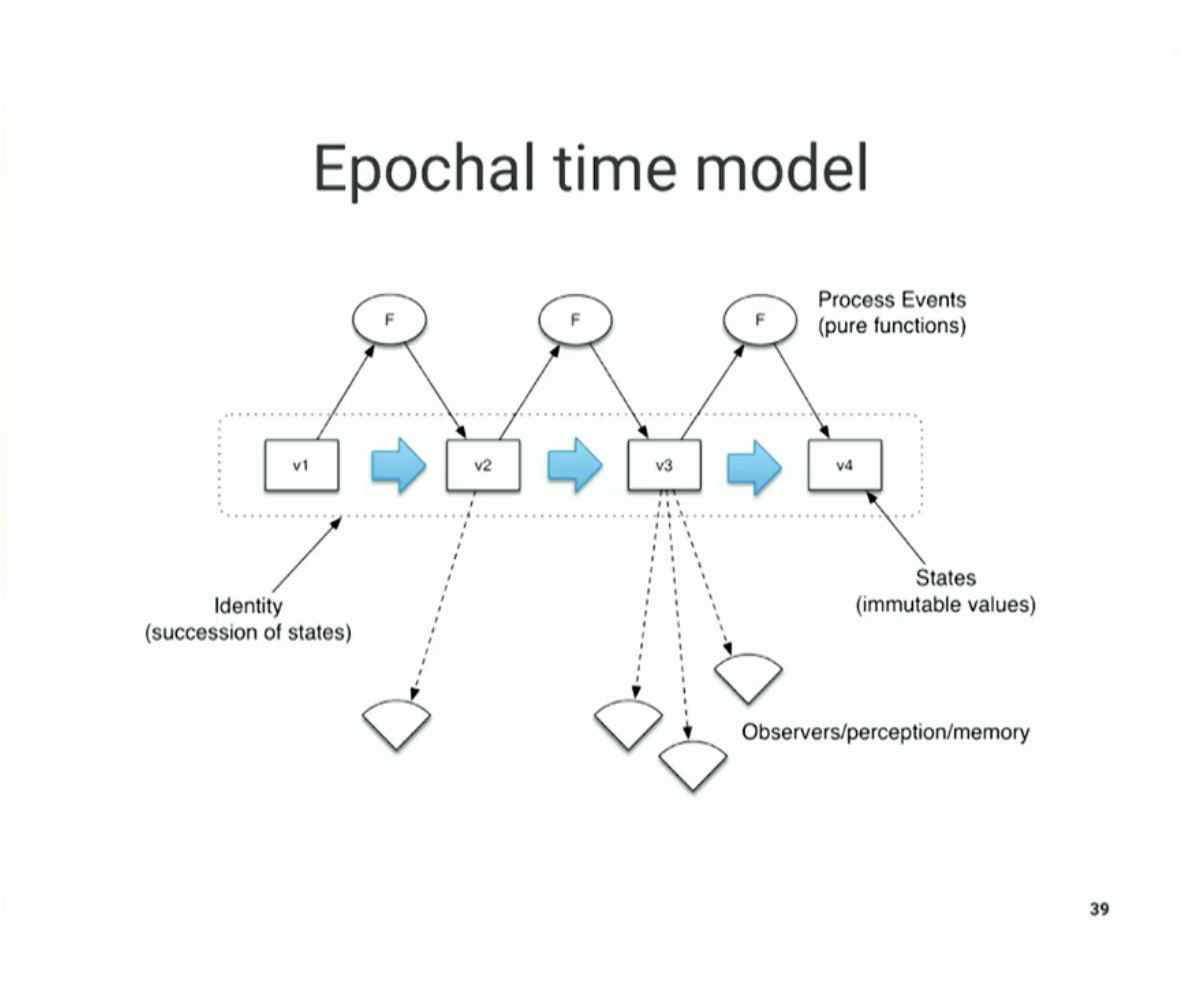

Figure 5: 01.02.36 Epochal time model

The other thing you need is some language for dealing with state transitions. And that is the epochal time model. I am obviously not going to get into this again here, but I have given talks about this before. So Clojure has this. And it was a combination of those things that let me say, "I think I have a reasonable answer for my friend". If he says, "How can I write a real program with this?", I can say, "Here is how you can write a real program, including a multi-threaded program, and not go crazy".

[Time 1:03:04]

```
slide title: Lisp - the good parts

+ dynamic
+ small
+ first class names - symbols, keywords
+ tangible
+ code is data
+ read/print
```

So there is lots of stuff I wanted to take from Lisp, and I think I talked about a lot of these. It is dynamic. It is small. It had first class names. It is very tangible. There is this code is data, and read/print, and I am going to talk a little bit more about that.

[Time 1:03:15]

```
slide title: R.E.P.L.

+ implies programmability in ways that 'interactive prompt' does not
+ read - human writable text -> data structures
+ eval - data -> executable code
+ print - data -> human-readable text
+ yes, can make interactive prompt but many other things
```

But there is the REPL. And I think that still people are like, "the REPL is cool because I get to try things". And that is true, but the REPL is *much* cooler than that. It is cooler than that, because it is an acronym. And it is cooler than that because *read* is its own thing. And what Clojure did by adding a richer set of data structures, is it made read/print into a super power. It was not just a convenience. It is not just a way to interact with people. It is not just a way to make it easy to stream programs around, or program fragments around.

It is now like, "Here is your free wire protocol, for real stuff". How many people ever sent edn over a wire? Yeah. How many people like the fact that they do not need to think that is a possibility. They can just do it? And if they want to switch to something else, you can. But it is a huge deal.

Eval obviously we know, it lets us go from data to code. And that is the source of macros, but I think again, it is much bigger than the application to macros.

And finally there is print, which is just the other direction.

[Time 1:04:25]

```
slide title: Lisp - Needs Fixing

+ built on concretions
  + (mutable) cons cell, lists
+ lists are kinda functional, read/print, other data structures are not
```

```
+ everything is not a list
  + lists are weak data structures
+ packages/interning
```

But Lisp had a bunch of things that needed to be fixed, in my opinion. It was built on concretions. A lot of the design of more abstractions and CLOS and stuff like that came after the underpinnings. The underpinnings did not take advantage of them, so if you want polymorphism at the bottom, you have to retrofit it. If you want immutability at the core, you need something different from the ground up. And that is why Clojure was worth doing, as opposed to trying to do Clojure as a library for Common Lisp.

The lists were functional, kind of, mostly by convention. But the other data structures were not. You had to switch gears to go from assoc with lists to a proper hash table.

And lists are crappy data structures. Sorry, they just are. They are very weak. And there is no reason to use them as a fundamental primitive for programming.

And also packages and interning were very complex there.

[Time 1:05:22]

```
slide title: Power - Strong Host Support

           power - ability to do

+ access everything
+ can reify interfaces
+ extend abstractions to host types
+ support host's abstractions
+ use host scalars
```

The other part about Clojure that is important is leverage.

Oh, I am running out of time. I am not going to talk about that.

[Time 1:05:30]

```
slide title: Large functional library built on abstractions

+ 600+ functions on
  + seq, collection, associative, indexed, counted, sorted, callable,
    deref, ref, lookup etc
+ defined with Java interfaces
  + did not ship with protocols
  + ClojureScript did later define same abstractions on protocols
```

Or that.

[Time 1:05:31]

```
slide title: The edn Data Model (Transit too)

+ Just the basics
+ Values only
```

```
+ Extensible
+ Good with names
+ The heart of Clojure
```

So the edn data model is not like a small part of Clojure. It is sort of the heart of Clojure. It is the answer to many of these problems. It is tangible. It works over wires. It is not incompatible with the rest of the world. Do other languages have maps? Associative data structures and vectors and strings and numbers? So it seems like a happy lingua franca. And why shouldn't we use the lingua franca in our program? Why should we have a different language? It is actually not that much better, and you have to keep translating.

[Time 1:06:04]

```
slide title: Static Types

[ photograph of an elephant ]
```

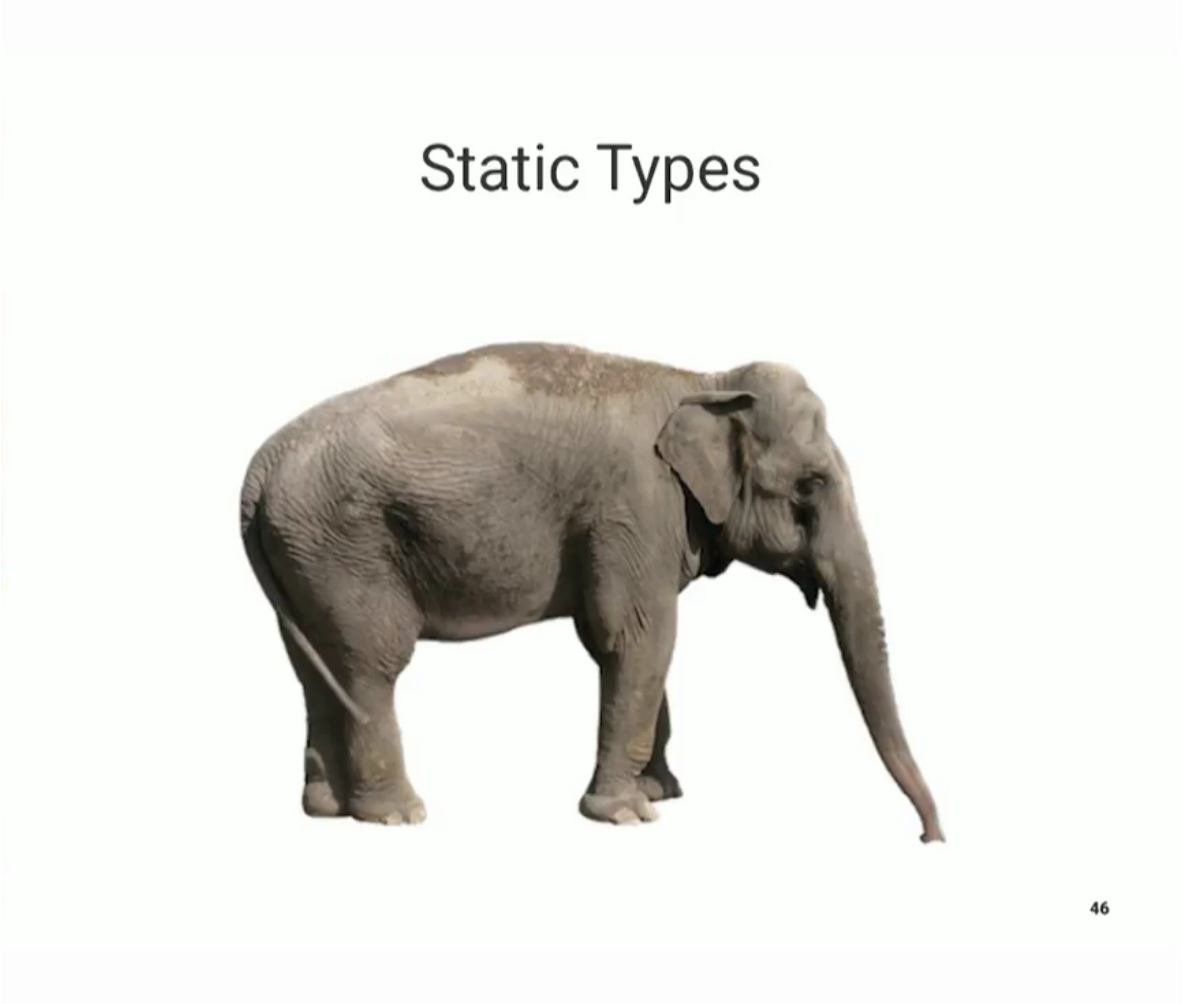

Figure 6: 01.06.04 Static Types

All right. Here is the final thing.

[Time 1:06:08]

```
slide title: The Joy of Types (SPJ)

+ Guarantee absence of certain kinds of errors
+ is a partial machine-checked specification
  + reverse :: [a] -> [a]
+ are a design language - types are UML of Haskell
+ support interactive development - intellisense
+ biggest merit: software maintenance
```

https://www.youtube.com/watch?v=brE_dyedGm0

[Link is to video of talk "Adventure with Types in Haskell" by Simon Peyton Jones]

Simon Peyton Jones, in an excellent series of talks, listed these advantages of types. Because this is the big thing that is left out of Clojure. There are no types.

They guarantee the absence of certain kinds of errors, which is true. And he would say – he does say – "This is the least benefit of static typing".

They serve as a partial machine-checked specification, and "partial" is the operative word here. It is very partial.

They are a design language. They help you think. You have a framework in which you can think about your problems.

They support interactive development like IntelliSense.

But the biggest merit, he says, is in software maintenance.

[Time 1:06:47]

```
slide title: But...

+ Not the big errors, testing still dominates real-world effectiveness
+ Names dominate semantics and always will -
  foobar :: [a] -> [a]
+ OmniGraffle > UML, design systems (db/wire) not just programs
+ Intellisense, perf optimization - yep, types help
+ Maintenance - overcoming coupling they create
```

And I really disagree with just a lot of this. It has not been my experience. The biggest errors are not caught by these type systems. You need extensive testing to do real-world effectiveness checking.

Names dominate semantics. a to a [in Haskell syntax: a -> a], list of a to list of a [in Haskell syntax: [a] -> [a]]. It means nothing. It tells you nothing. If you take away the word "reverse", you don't know anything. You really don't. And to elevate this to say, "Oh, this is an important thing. We have all these properties". It is not true. It just is not true. There are thousands of functions that take a list of a and return a list of a. What does that mean? It means nothing.

And checking it ... I mean, if you only had a list of a-s, where are you going to get something else to return? I mean, obviously you are going to return a list of a-s, unless you are getting stuff from somewhere else. And if you are functional, you are not.

How many people like UML? How many people have ever used a UML diagramming tool? It is not fun, right? It is like, "No, you cannot connect that to that". "Oh no, you have to use that kind of arrow". "No, you cannot do this". "No, you cannot...". It is terrible. OmniGraffle is much better. Draw whatever you want. What are you thinking about? Draw that. What is important? Write that down. That is how it should work.

Yes, IntelliSense is much helped by static types. And performance optimization, which he did not list, but I think this is one of the biggest benefits. We love that in C++.

And maintenance, I think it is not true. I think that they have created problems that they now use types to solve. Oh, I pattern-matched this thing 500 places and I want to add another thing in the middle. Well thank goodness I have types to find those 500 places. But the fact was: that thing I added, nobody should have cared about about except the new code that consumed it. And if I did that a different way, I would not have had to change anything except the producer and the consumer, not everybody else who could not possibly know about it, right? It is new.

[Time 1:08:47]

```
slide title: Puzzles vs Problems

A logical theory may be tested by its capacity for
dealing with puzzles, and it is a wholesome plan, in
thinking about logic, to stock the mind with as many
puzzles as possible, since these serve much the
same purpose as is served by experiments in
physical science

-- Bertrand Russell
```

So I mean for young programmers, if everybody's tired and old, then this does not matter any more. But when you are young, you have got lots of free space. I used to say "an empty head", but that is not right. You have a lot of free space available, and you can fill it with whatever you like. And these type systems, they are quite fun, because from an endorphin standpoint, solving puzzles and solving problems is the same. It gives you the same rush. Puzzle solving is really cool.

[Time 1:09:26]

```
slide title: Spec

+ Clojure is dynamic for good reasons, not ease
+ Specs and proof should be a la carte
  + diversity of needs, models, costs
+ Pointed at system-level (prog/wire/db)
+ Next iteration will increase its programmability
  + producing and consuming
```

But that is not what it should be about.

I think that this kind of verification and whatnot, it is incredibly important, but it should be a la carte. Depending on what you need to do, depending on the amount of money you have to spend, depending on what you want to express, you should be able to pull different kinds of verification technology off the shelf, and apply it. It should not be built in. There is a diversity of needs. There is a diversity of approaches of doing it, and a diversity of costs.

In addition, I think to the extent these tools can be pointed at the system level problem, and not some language parochialism, you get more bang for your buck. How many people have used spec to spec a wire protocol? Yeah. There is going to be a lot more of that going on. And I will not talk much more about spec, but the next version will increase programmability.

[Time 1:10:12]

```
slide title: Information vs Logic

+ We can't explain how to drive a car or play Go
+ Information-trained brains
+ Human-written bodies (manipulable programs)
  + Via which brains _see_ and _act_
+ Real-world safety is going to come from experience, not proof
```

So finally, information versus logic. The bottom line is, "Where are we going in programming?" The fact is, we actually do not know how to drive a car. We cannot explain how to drive a car. We cannot explain how to play Go. And then, therefore we cannot apply traditional logic to encoding that and make a program that successfully does it. We just cannot do it. We are approaching problems in programming now that we do not know how to do. We do not know how to explain how to do. Like, we know how to drive a car, but we do not know how to *explain* how to drive a car.

And so we are moving to these information-trained brains. Deep learning and machine learning, statistical models and things like that. You use information to drive a model that is full of imprecision and speculation, but that is still effective because of the amount of data that was used to train it at making decent decisions, even though it also could not explain necessarily how it works.

These programs, though, are going to need arms and legs and eyes. When you train a big deep learning network, does it get its own data? Does it do its own ETL? No. It does not do any of that. When it has made a decision about what to do, how is it going to do it? Well, when we get to Skynet, it will not be our problem anymore.

[Audience laughter]

But for right now, it is. And I think it is quite critical to be working in a programming language that is itself programmable, that is amenable to manipulation by other programs. It will be fun to use Clojure to do brain building. But it will also be useful to be able to use Clojure for information manipulation and preparation, as well as to use Clojure programs and program components as the targets of action of these decision-making things.

In the end, real-world safety is going to come from experience. It is not going to come from proof. Anybody who gets on stage and makes some statement about type systems yielding safe systems, where "safe" mean real-world? That is not true.

[Time 1:12:29]

```
slide title: The Problem Tackled by Deep Learning

[ Same text as earlier "The Problems of Programming" slide, but now
only the "Misconception" box is highlighted in a different color. ]

Domain Complexity
_Misconception_

10x
```

```
PLOP - place oriented programming
Weak support for Information
Brittleness/coupling
Language model complexity
Parochialism/context
weak support for Names
Distribution
Resource utilization
Runtime intangibility
Libraries
Concurrency

10x

Inconsistency
Typos
```

So, this is what is really interesting. Deep learning and technologies like that are pointed above the line, above that top 10x. They are pointed at the misconception problem. They say, "You know what? You are right. We do not know how to play Go. We do not know how to drive a car. Let us make a system that could figure out how, and learn it, because otherwise we are just going to get it wrong".

[Time 1:12:56]

```
slide title: Programmable Programs

+ Generic information representation/emphasis
+ Compose args without adopting/understanding elaborate type system
+ Dynamic discovery and invocation
+ Dynamic enhancement
```

So, I am going to emphasize that we write programmable programs, and that Clojure is well-suited to that. We have a generic way to represent information and emphasis. We have a generic way to compose arguments without adopting the type system. It is hard enough to drive a car. If you have to understand monads too, it is just not going to work.

A reified system is subject to dynamic discovery, and I think spec combined with the rest of Clojure being reified is a great way to make systems that other systems can learn about, and therefore learn to use. And of course we have the same ability to enhance our programs over time.

[Time 1:13:38]

```
slide title: Be Effective!

+ Point your programs at the world
+ Logic is your tool, not your master
+ Design at the system level, outside of language
+ Embrace the opportunities of functional dynamism + brain-building
  + Make programmable programs
+ Solve problems, not puzzles
```

So, I would encourage you all to embrace the fact that Clojure is different. Don't be cowed by the proof people.

[Audience laughter]

Programming is not a solved problem. Logic should be your tool. It should not be your master. You should not be underneath a logic system. You should be *applying* a logic system when it works out for you.

I am encouraging you to design at the system level. It is not all about your programming language. We all get infatuated with our programming languages. But you know what? I am actually pretty skeptical about programming languages being the key to programming. I do not think they are. They are a small part of programming. They are not the driver of programming.

And embrace these new opportunities. There are going to be a bunch of talks during the conference about deep learning, and take advantage of them. Make programmable programs and

[Time 1:14:34]

```
slide title: fin

The true function of logic ... as applied to matters of
experience ... is analytic rather than constructive;
taken a priori, it shows the possibility of hitherto
unsuspected alternatives more often than the
impossibility of alternatives which seemed prima
facie possible.  Thus, while it liberates imagination as
to what the world may be, it refuses to legislate as to
what the world is

-- Bertrand Russell
```

solve problems, not puzzles. Thank you.

[Audience applause]

[Time 1:14:39]