The Value of Values

- **Speaker: Rich Hickey**
- **Conference: goto; conference 2012 - May 2012**
- **Video: https://www.infoq.com/presentations/Value-Values**
- **Slides: http://gotocon.com/dl/goto-cph-2012/slides/value-of-values.pdf**

[Time 0:00:00]

```
slide title: The Value of Values

Rich Hickey
Datomic, Clojure
```

Thanks very much for having me. Today's talk is going to be: The Value of Values.

[Time 0:00:12]

```
slide:

I.T.
```

I would like to start by polling the room. How many people are in I.T. or an I.T. related field?

[Audience laughter]

This is great. The key is to start with an easy question after the party.

What does that stand for? It stands for Information Technology. And one of the themes of this talk is going to be: keeping in mind what information means, and what we are actually trying to accomplish. And looking at the tools and technologies we are using, and seeing if they are actually suitable for accomplishing what we are trying to do.

[Time 0:00:57]

```
slide title: Information

+ _Inform_
  + 'to convey knowledge via _facts_'
  + 'give shape to (the mind)'
+ _Information_
  + the facts
```

So we will start with that key word, because the technology part I think is straightforward. And we will look at information. And of course we will start with a definition. Everybody knows this is my schtick. If you are going to do a talk, you just pick a word, then you go look it up in the dictionary, and you are rolling. It is a cheap trick, but it is actually quite useful, because there is a lot of the history of human thought sort of boiled down into language.

So if you look at the word "information", it is based on the word "inform", and "inform" actually means to convey knowledge via facts. To shape your mind, or to shape someone else's mind, by communicating facts to them. That is what it means to inform.

And the key word I think in here, which is going to be a theme of this talk, is "facts". Because we are going to try to give more precise meaning to that, and see if our information technology actually manipulates information. Because that is what information is. Information is those facts that we use to inform. And not anything else. Not any of the artifacts we use to represent it.

[Time 0:02:14]

```
slide title: What is a Fact?

+ _Place_ where specific information is stored
+ There is a place for every piece of information
+ Facts have _operations_, e.g. _get_ and _set_
+ Operations control how facts can _change_
+ To convey a fact, convey its _location_

[ The entire slide has superimposed over it a big red circle with a
red slash through it, used in signs to mean "No". ]
```

So start again with: what is a fact? This is not the dictionary definition.

So a fact is a place where information is stored. And there is a place for every piece of information. And every fact has a set of operations, like definitely get, and maybe set, although if set does not have the right controls for the fact, there might be other kinds of operations that would control that.

And then that is essential, that operations control how facts can change. And then when we want to communicate about facts, all we need to do is convey their locations.

Right?

How many people are uncomfortable right now?

[Audience laughter]

I am. I cannot even keep a straight face with this slide. Are you kidding me? This is not right. This is very, very wrong. If my partner Stu Halloway is in the audience, he probably almost had a heart attack while this slide was up.

This is *not* what a fact is. And yet, a lot about that description is similar to what our programs do.

[Time 0:03:37]

```
slide title: Place

+ 'A particular portion of space'
+ 'An area used for a particular purpose'
+ Memory address, disk sector
```

So let us dig in to the word "place". And "place" means a particular portion of space. And space is another word that is very interesting, and that is going to come up later. The key word here is "particular", and "portion", and the delimiting nature of this.

This other definition has that same characteristic. An area used for a particular purpose. A specific area.

And we are really comfortable with the notion of place, because we have two critical places we are constantly manipulating with our programs. One is memory, and the other is the disk. And they very much co-align with place. They are places.

There is only so much memory, and there are particular addresses in memory. And there is only so much disk space, and there are sectors on the disk. And these are all places. They are subdivisions of the universe. There is only so much of the universe that is on my hard drive. There is only so much of the universe that is in the memory of my computer.

[Time 0:04:39]

```
slide title: 'Information' Systems

+ In memory
  + mutable objects are abstractions of places
  + objects have _methods_
+ In durable storage
  + tables / documents / records are places
  + DBs have _update_
```

I want to look at what we are calling "information" systems now. Because we are building information systems. And in memory we are building them out of mutable objects. But mutable objects are actually abstractions of places. They do not actually have meaning other than that. They are little barricades we have set up in front of memory, so that we do not have to directly manipulate memory addresses any more. So we have this abstraction that is an object, that helps us manipulate that place without too much craziness.

And a key characteristic here is that objects have methods. They have those operations we talked about before, that facts really do not have. Objects critically have them. They are operationally defined. And we use them to provide a layer of abstraction over the places that our program uses.

And the same thing happens in storage. We again have tables, and documents, and records, and these higher level notions that fundamentally are born of the desire to, again, abstract away the details of the fact that we are working with a place. But the abstraction is not really a first class abstraction, *other* than to hide place from our programs. It actually is not a sound abstraction above that.

And one of the ways you can tell it is a place oriented abstraction is this update, this same notion of going to a particular part of the universe and manipulating it. So these are what we are building information systems on right now, and I think we may have some difficulty seeing how that is correct.

[Time 0:06:29]

```
slide title: PLOP

+ PLace-Oriented Programming
+ New information _replaces_ old
+ Born of limitations of early computers
  + small RAM and disks
+ Those limitations are long gone
```

So I have a new label for this. It is called PLOP, and PLOP is place-oriented programming. And it is what most of us have done for most of our careers, and most of us continue to do.

And it is characterized by a very basic operation, which is: new information replaces the old information. It is that simple. If that is happening, you are doing place oriented programming. It does not matter if it is in memory. It does not matter if it is on disk. If new information replaces the old, you are doing place oriented programming.

And it does not matter if your implementation technology is not actually doing that directly, so I do not care if you are using MVCC, or an append-only database. If the logical result is that new information replaces the old, that is an in-place system, even if for efficiency it appends on the disk. If, in the end, it can only give you the most recent piece of information, that is a place-oriented system. It does not matter if it is actually going back to the same disk sector.

And there is a very good reason why we were doing place-oriented programming decades ago. The first computers were really tiny. They had incredibly limited memories, very, very small disks, if they had any disks at all. And so we *had* to do place-oriented programming. There was no way to get a computer to do anything useful unless we took the tiny amount of memory that we had, and completely defined our program in terms of the role of every place of memory, and helping our program accomplish what it was supposed to accomplish.

[Time 0:08:13]

Guy Steele gave a talk in the most recent couple of years where he had this great anecdote about the worst program he had ever written. And he held up a card showing what it was. And then he just described the computer on which it ran, which had 4000 words of memory. And how there was this map of that memory, and this part here was this dispatch table. And then there was some code here, but sometimes you could cram data into it. Then there was this other jump table, and then some data structures. And every program knew exactly where in memory those portions were, and directly used the addresses. That is how you had to do it.

And then we got bigger memories and we said, "Ah, we do not really want to program with addresses any more". And so we added some stuff, but the basis for the way we computed was still that. We are still just trying to do that, but probably not deal with the hassles of knowing the addresses directly. We use some indirection there.

The problem is that those constraints, the constraints Guy Steele and the early pioneers of computing faced before him, they are gone. Computers, just in the time I have been using them, are a million times more capacious in memory and disk. Than when *I* started, which was after he started.

But we are still doing place oriented programming. I think we definitely need to consider why that is.

[Time 0:09:43]

```
slide title: The Efficiencies of Place

+ Ok, when 'birthing' new values
  + birthing == prior to perceptibility
  + i.e. prior to becoming a fact
+ _But_: an _implementation_ detail
+ I.T., not T.T.
```

So one reason that always gets brought up right away is that there is efficiency to manipulating places. And that is definitely true. I am not opposed to that. I have bashed as many bits as the next person. And I know how much fun that is, and I know how fast that can be.

And I think that there is still a role for that, and there will always be a role for that. And one way to talk about when that is appropriate is to have a notion of this birthing process, a point in time when you are starting to create a new value, and we will talk more precisely about values in a minute. And in setting up that value, you need to manipulate memory, for instance. You need to manipulate places.

That is completely OK. I would never advocate languages that did not, for instance, let you manipulate the contents of an array. Because during this process, you need to be able to do that in order to write efficient programs.

But this birthing process is a window that ends, and it ends whenever the thing that you have made is going to become visible to any other part of your program. At that point it has become a fact. It has become perceptible. And then you have to stop doing place-oriented programming. Because as we will see, it is not a fit for the models we are trying to build.

So this use of place to create values, or the use of place to represent values under the hood, is an implementation detail.

Of course we have to use places. Our computers have memory and they have disks. But what is important is that our program is not *about* places. It is information technology. It is not "technology technology". We have taken abstractions of the technology, and raised them up to being what the program is about, and that is an error.

There was a reason why we had to do it, but we do not any more.

[Time 0:11:51]

```
slide title: Memory and Records

+ We've co-opted
  + and believe our own mythos
+ Mental memory is _associative_ and _open_
+ Real records are _enduring_
  + and _accreting_
  + not erase and overwrite
```

So two words that I think are very important are "memory" and "records". We have to remember: these words had meanings before we started to try to emulate them with tiny computers. We used these words for millennia prior to that.

And we have not only co-opted them, but I think that we are starting to believe our own myths about what memory and records are. They are what our programs say they are, as opposed to what they *really* are.

Real memory is a cognitive abstraction over how our brains work. And some of the characteristics about it that are really interesting are the fact that it is associative. If your friend gets a new phone number, it does not go into your brain and find the phone number neurons for your friend, and overwrite them with the new phone number. That is not how memory works.

You get the new phone number. It is some novelty. And your brain accommodates it, and what was there before. So it is associative. There is some connection between my friend and the phone number, and those numbers.

But it is also open. It is not a place. Your friend's phone number is not a place in your brain. And memory is about that activity, acquiring your friend's phone number when he changes it.

And record keeping existed before we had computers. Records are enduring. People did not go back to their parchments and scrub them out when there were new facts. They did not go back to their stone tablets and pave them over with concrete and then re-etch them. They just wrote new pieces of paper, and carved new stones.

They are enduring, so we keep them around. And they are accreting. If you have new information in the old record keeping systems, you added it to what you had already. You did not go and erase it.

So these are critical notions. We actually do pretend that our systems do this work. But to the extent that we are using memory and records the way we have become accustomed to, we are not actually.

[Time 0:14:10]

```
slide title: The Point

+ Values have many advantages
  + in process
  + across processes
  + in storage
+ We _know_ these things
+ _Place has no role in an information model_
```

So the point of this talk is that values have many advantages over this place oriented programming. And I am going to talk about values in many different ways, in order to try to give you a better idea of the many kinds of meanings it can have for programming.

In particular, though, I want you to focus not only on values in memory and functional programming, whatever you think of when somebody says you should program with values. But also our use of values in communicating across processes, and our use of values inside storage systems, because there are many architectural advantages to values that go beyond the parochial notion that a program might have.

The other point of this talk is: you already know this stuff. You were all made uncomfortable by that first slide. Your activities show that you know this stuff. The only thing that counteracts the fact that you know this stuff is the fact that you continue to choose, some of you, technologies that do not implement what you know to be true. And there may be many reasons for that.

But the *most* important point of this talk is that place itself has no role in an information model. It is *only* an implementation detail. If you elevate place to be a first class thing in your "information model", it is only an "information model". It is pretend. It is not actually doing its job.

[Time 0:15:41]

```
slide title: Value

+ 'Relative worth'
+ 'A particular magnitude, number or amount'
+ 'Precise meaning or significance'
```

So let us dig in to the word "value". This is a particularly tricky word, and the title of the talk is a little bit tricky. "The Value of Values" seems to imply two meanings of the word "value".

The first one is "relative worth". "The *Value* of Values". "How do you estimate the worth of something?" is a notion of "value".

And then we have what is probably the clearest mapping to programming of the word "value", which is: a particular magnitude, or numeric value, an amount. This is "42". This is the one we can really hang on to. Everybody understands 42 is a value.

But it ends up that these two definitions are *not* different, when you take this third definition into account, which is: precise meaning or significance. Because what ends up happening is: all notions of value are about being able to directly perceive something and compare it to something else. And we will see that that allows us to have a broad notion of value, which will not only cover 42, but other things that we encounter.

[Time 0:16:52]

```
slide title: Is a String a Value?
```

```
+ Is it immutable?
+ Equality, comparability are basis for logic
+ Who wants to go back to mutable Strings?
```

So what else might we encounter? Strings. Are strings values? How many think strings are values? How many think strings are not values?

It ends up that the answer to this question is a question. It depends on your programming language. Are strings immutable in your programming language? If they are, then strings are values. If they are not, then they are not.

And how many people work in a language where strings are mutable? How many people have *ever* worked in a language where strings are mutable? How many people worked in a language where strings were mutable, and now they are not in the language they work in now? There have to be some people who programmed in C, and then in Java, yeah?

Of people who have programmed in languages that had mutable strings, and then ones that did not, how many people want to go back to mutable strings? Wow. Do you work with other people?

[Audience laughter]

It is really tricky, because we have accepted, at least in Java. How many people program in Java here? So in Java, we sort of accepted string as a value. We have moved on from 42. We said, "Oh, no, this composite thing that has a bunch of different parts, a string, could be a value." And it ends up that it is.

If it is immutable, it now taps into that definition of value we saw before. Because by being immutable we can go and take a string value, and another string value, and say: are they the same? Do they have the same magnitude? Are they talking about the same thing? Are they expressing the same specific meaning? All of those definitions of values apply to something that is not mutable. So that relative worth thing kicks in.

And I do not thing anybody who has programmed with both wants to go back. I am not actually sure I believe you.

But by and large, this is something that we have accepted.

[Time 0:18:56]

```
slide title: Programming Values

+ Immutable
+ Don't _need_ methods
  + I can send you values without code
  + and you are fine
+ Are semantically transparent
+ Can be abstracted
```

So if you want to expand the notion of value up, and talk about programming values, we are going to have some characteristics we really care about. The first and unconditional one is: that they be immutable. We are going to see, as things become mutable, our ability to do any of the things that we say we can do with information and values, disappears.

On the other hand, another important characteristic of values is that they do not need methods. Now I am not saying that values cannot have methods. I am not saying you cannot have an object in your

programming language that has the role of a value, and meets the criteria for value, and has methods. I am not saying that that is not allowed.

But the important thing about values is that they do not *need* to have methods. They are not operationally defined. If I can convey a value to you somehow, and I have forgotten to give you any code, you can use it because semantically the value is accessible.

So that is the other critical thing. It must be immutable, and it must be semantically transparent. There cannot be any operational interface over a value that tries to encapsulate what it means, or your ability to do equality on it. You might have additional methods. You might have toUpper on a string. That is just sort of object oriented goofiness, but it is harmless in this case.

The important thing, though, is: you cannot have a value where only on Tuesdays, by calling this method and that method, can you see what it is about. They have to be semantically transparent.

And it is OK to have abstractions. In particular, when you start talking about composites and collections as values, you will often have an abstract definition of that. But that abstract definition satisfies the other two critical properties: it is immutable, and it is semantically transparent. The abstraction is not trying to get in the way of you seeing what it is, at seeing *all* of what it is. It may just be hiding the storage part.

[Time 0:20:47]

```
slide title: Values Can be Shared

+ Share freely
  + aliases are free
+ No one can mess you up
  + nor you them
+ Incremental change is cheap
+ _Places_
  + Defensive copy, clone, locks
```

So let us go through some properties of values, and how they compare to places.

The number one property is: values can be shared. And they can be shared freely. And that the way you share them is just by aliasing them. Because you know that they are immutable, if you ever encounter a value, you can just start using it.

And it is funny, because people talk about functional programming, and higher order functions, and all of this stuff, and concurrency and other advantages. But when someone actually goes from not using a functional programming language to using one, one of the deepest pleasing benefits they have is *this one*. It is the fact that when you program with values, you can share pervasively, and you *never* need to think or worry for one fraction of a second.

You cannot mess anyone else up, and they cannot mess you up. All values are freely shareable. If you have never done it before, it will change the way you program forever. It really makes a big difference.

One of the things that also happens, especially when your values are implemented with persistent data structures, is that incremental change is cheap. So it is quite common to say, somebody gives you this big thing, and you are like, "I love that big thing, except for the first thing. I would like to have that big thing except for the first thing." And that ends up being completely straightforward to (a) do, and [b] inexpensive to do.

So that is really great. Now if we compare that to programming with places, what happens? Defensive copying. How many people have heard the term "defensive copy"? Why do we need to defend ourselves from ourselves? This is really not a great phrase to be using every day.

Cloning. Another nasty notion. And locks. These are all things that are either part of, or in the way of, sharing when you are programming with places.

[Time 0:22:35]

```
slide title: Reproducible Results

+ Operations on values are stable
+ Testing
+ Debugging
  + reproduce failures w/o replicating state
+ _Places_
  + must establish matching 'state' first
```

Reproducible results are another fantastic benefit of values, because operations on values are stable. You do them over and over again. They never give you a different answer.

This is really a great benefit when you are doing testing, obviously, because if you want to say, "it still works", hopefully you have code that is reproducible in the first place.

And you actually spend a fair amount of time with place oriented programming making that sentence true, that the test actually, when run twice, *should* return the same result. Where if you are programming with values, that is not even a question.

Debugging is also critically different when your program programs with values, especially when your architecture is based around values. So some customer has a problem in the field, and you have a value oriented program, you can say, "obtain the value from your database and the query you were running, and email them to me." Just the value that was the input to the process, and just the query, those two things, and I can reproduce here over email.

Versus what? How many people have ever tried to set up a database, and a running process, that emulates a customer failure? That is not a party, right? Not fun. Not fun.

And that is the problem with places. You have this sort of global state that you have to reproduce in order to debug a field problem. That is very very tough.

[Time 0:24:00]

```
slide title: Easy to Fabricate

+ Anything can create compliant values
  + for testing, simulation
+ _Places_
  + must emulate operational interface
```

Another advantage of values is that they are easy to fabricate. Anything can create a value. Any programming language can make a value. You may have written it in this, and then later you need to have somebody who uses different languages drive it to see if it is working.

So for testing it is really fantastic that you can fabricate inputs to test programs, using any technology. You do not have to sort of: get the class library that has the right classes and interfaces related to the gook that you used in your program.

It is like: your program takes data. You now can write another program that can produce data to test it.

And also for simulation purposes. So when you start raising your testing up to the next level, and you are trying to drive your program to different kinds of situations, all you need to do is algorithmic generation of data to get a variety of simulation points for your program.

If your program can only get into a particular state by a series of interactions through objects, how are you going to algorithmically drive that program to different kinds of test cases? It is a huge problem. It is just a mess. Whereas if you can just algorithmically generate data, you are done.

And again it goes to this point about places. With places you have to emulate an operational interface, and that is a ton more work. And also when you want to drive it, you have to drive it through the operational interface instead of with data.

[Time 0:25:30]

```
slide title: Thwart Imperativeness

+ Values refuse to help you program imperatively
  + That's a feature
+ Imperative code is inherently complex
+ _Places_
  + Encourage and require imperativeness
```

Imperativeness. We love it, right? And values are in the way. That is a feature. That is not a negative aspect. They just refuse to help you do this.

And I think that once you start using languages that make values the default, you feel frustrated, initially, about this. But in the end, it is a tremendous benefit. Because imperative code is just more complex, as used to it as you may be. It is more complex.

And the problem with places is, they *force* you to do this. It is the exact opposite. Values thwart you, and places force you, to write imperatively, and therefore in a more complex way.

[Time 0:26:23]

```
slide title: Language Independence

+ Pure values are language independent
  + _the_ polyglot tool
+ _Places_ are defined by language constructs (methods)
  + can be proxied, remoted, with much effort
```

Starting to lift the game a little bit out of your local view, which might be, "I am in Java, and I am doing this. I have this interface. I have this class model, and blah blah blah." A great thing about values is language independence. If you ever want to pretend you are a polyglot shop, you are going to immediately face a challenge with all of your interface driven, object driven designs, which is: you can have them all over in your Java program, but then your Python program, or your JavaScript program, it does not know how to talk about that stuff. It has no means of doing it.

And immediately you are going to face this pressure to move away from that. And towards what? Towards values is where you are going to end up. They are the tool for polyglot programming. They are the tool that gives you this independence in language. Because places are defined by language constructs, you are stuck. You are really stuck. You do not have a definition independent of your language that you can use as a basis.

And sure, you can build proxies. You can automatically build SOAP interfaces to your objects, and remote your objects, and generate matching objects in different languages, but that is just a ton of effort. And it is not really adding any value.

[Time 0:27:43]

```
slide title: Values are Generic

+ Representations in any language
+ Few fundamental abstractions
  + for aggregation (lists, maps, sets)
+ _Places_
  + Operational interface is specific
  + More code
  + Poor reuse
```

So this language independence actually falls out of a bigger property, which is that they are generic. We can get representations in any language, as we said. But the other thing is that there are very few values, in the general sense. Once you start programming with values, you do not end up with a lot of specificity. There is a logical notion of a list. There is a logical notion of a map, and a logical notion of a set, and strings and numbers and whatever. But you can probably exhaust what you need to use in the value space with fewer than 20 of these things.

Whereas how many people can build a system with 20 Java classes? Just 20? No large system, right? As the system gets larger, how many more classes do you need? More and more and more and more and more. They keep going on and on and on.

And that is because operational interfaces are specific. That generates a ton more code. And it actually is a counter argument to the promise of object oriented programming. One of the promises was reuse. That is the big lie of object oriented programming. Every new thing you have to do, you write a new class. Where is the reuse in that? There is none.

The other thing is, you are sort of breaking away from the job you are trying to do. If you are trying to represent information, you need to represent facts. You need to have values in order to have things be comparable. If I have a person class, and you have a person class, in their own namespaces, and they have name, address, and email, and name, address, and email, what can we do with those two things? Nothing! Even though they are semantically identical, they use the same names, and they use the same names for the fields, they are completely not interoperable. Even if they all had public getters. They are complying with the accessibility part. The specificity that you added killed your reuse.

[Time 0:29:44]

```
slide title: Values Are the Best Interface

+ For subsystems
  + can be moved
  + ported
  + enqueued
+ _Places_
  + application, language and flow coupled
```

And again, getting more in the large, or looking towards programming in the large, values make the best interface. This is actually one of the biggest problems I think we have right now. Is that when we

are working in the small, we say, "we are going to have this new thing". And we start with a monolithic design, but within that design it is not monolithic. We have a subsystem for this, and a subsystem for that, and a subsystem for this.

That is all great. And then it is like, "Oh, you know what? That is getting too big for this box. I want to move this out of that box to this other box." And when I do that, I think there is a different programming language that would make that easier, or it is going to be shipped to another team that works in a different programming language, and so we are going to do this other thing.

If you have a value based interface, you can do that move. If you have programmed with data driven interfaces, you can do that move. You can port that code. Or you can write new code that interoperates in a different language, because it is data driven.

Another critical thing you can do if you have a value oriented interface is: you can enqueue it. So even if you stay in process, a lot of times one of the architectural needs you have is: you know what? I am calling this, and calling this, and calling that. And I need to buffer. I need to do some more management of things. Or maybe I want to get some more concurrency in play. And therefore I would like to enqueue those calls. So I want to set up a queue. So now I have this flow, and maybe get some pipelining in my program.

[Time 0:31:15]

If you have called this specific interface, called this specific interface, and called this specific interface, and then you want to pipeline that, what can you do? You are stuck! Because you have got to go and build like proxies that look like your objects, that then have a queue inside, that then spit out on the other end another thing that looks like what it was talking to. And heaven forbid it was bidirectional. You are just totally toast.

But if you had a data driven interface, like this guy was calling that guy, but it was just passing data. If you want to stick a queue in the middle of that, that is straightforward to do, because you can put values on queues.

So in contrast, if you are doing place oriented programming, your stuff is application specific, your stuff may be language specific, and it may be coupled to your program flow. Architecturally, you are dramatically limited. And this is a big deal, because you *desperately* need to be able to take your small programs, and make them large programs, and take your one machine programs and make them N machine programs. If you cannot start with an N machine program, you are not forced into this.

But the thing is, we know this. Because when we program in the large, we do not pretend we have objects. We do not create operational interfaces. We do not chat. We do not use CORBA any more. That is dead. That lost, for good reasons.

When we actually start out building a more distributed system, we program with data all of the time. We already know how to do this. We use data on the wire. We use RESTful interfaces. Everything is different in the large. Why are we stilling doing this arcane goofy memory abstraction oriented stuff in the small? It does not map to the large. It is not going to help us make our programs bigger. And there are no benefits to it.

As soon as we look at our programs in the broader sense, we don't do this. We don't make the same choices in the large. We are still making them in the small. I think it is just because we are comfortable with our programming languages.

[Time 0:33:08]

```
slide title: Values Aggregate

+ Values aggregate to values
```

```
+ So all benefits accrue to compositions
+ _Places_
  + Combinations of places, what properties?
  + Need new operational interface for aggregate
```

Another key advantage of values is that they aggregate. In particular, values aggregate to values. So if I have five values, and I put those five values in a value list, that resulting thing is a value. In particular, everything I have said about values accrues to that composite. That composite thing has all of the advantages of a value that all the value parts of it have. It is transparent. It is transmissible. All of the characteristics are great.

Now contrast that with programming with places. If you have a bunch of mutable objects, and you combine them into a bigger thing, what properties does it have that you can understand? Even if you really understood all of the sub components? What properties does the composite have? None!

You have to start from zero again, defining the operational interface of the aggregate, even if you had very carefully defined cloning, and copying, and locking policies for each part, as soon as you combine them together, you are toast. None of those things work. You now no longer have a copying policy, no longer have a cloning policy, no longer have a locking policy on the aggregate. So *nothing* composes with places. That is a big negative.

So now I would really like to start broadening the notion of what we are talking about, when we are talking about values, to outside one process. To talk about them in the large, and in the small. Still mention the others.

[Time 0:34:39]

```
slide title: Extended Value Propositions

+ Mechanism for conveyance and perception
+ Mechanism for memory
+ Reduced coordination
+ Location flexibility
+ _Essential_ for decision making
```

And talk about a few what I will call extended value propositions.

Using values as a mechanism to convey things and to perceive things. Using values as a mechanism for memory. How values will reduce coordination. How they provide location flexibility. And finally, how they are essential to making programs that support decision making, which is our job in I.T.

[Time 0:35:07]

```
slide title: Conveyance

+ In the small
  + Aliases of values convey value
  + Mutable things on queues convey nothing
+ In the large
  + Values rule on the wire
  + No reproducible values in PLOP DBs
```

So we have conveyance. And conveyance means: to send something to somebody else. So this is sending.

In the small, with values it is really straightforward. If I give you any reference to the value, I am done conveying it to you. I have conveyed it. It is extremely cheap. And again, as we saw before, it is worry free.

Imagine, though, that you want to try to do conveyance with places. So you have this mutable object, and you put it on a queue, and later somebody is going to consume that queue. What actually have you communicated to that person? Nothing! You do not know. You put it on the queue now. It is just a reference to a thing that could change. Whatever your intent was in conveying it, it is not captured by that mutable thing on a queue. So conveying places is an extremely difficult thing to do.

We waste a ton of time. Everybody is thinking about places. You know I do these things, but I spend a huge amount of work trying to do them. I have to try to clone it, or something like that. You have to turn it into a value, essentially.

Now look at conveyance in the large. Again, here I think we figured this out. Values rule on the wire. We do not really do anything other than values on the wire now. HTTP, really all distributed programming, puts values on the wire. We do not set up multiple objects with tiny little interfaces and chat across the wires. We just do not do it.

People imagined that. When they first started, objects were all the rage. They were like: distributed objects, because that is all we can think about, so we will think about it broadly. But it was an utter complete total failure. And we are done with it. Again, in the large we understand this. So that is sort of the wire part.

And then in databases, we have the same problem. If I give you the primary key of a record in the database, if I send that to you over a queue, what have I actually communicated to you? Nothing! Because what you are going to see depends on when you look that thing up, just like with objects before. Putting an object on a queue, sending somebody the primary key of something, if what is behind that stuff is places, you actually have not conveyed anything specific. In other words, you have not conveyed information.

[Time 0:37:23]

```
slide title: Perception

+ In the small
  + Values: to reach is to perceive
  + _Places_: How to perceive a coherent value of object with multiple getters?
+ In the large
  + Values still rule on the wire
  + No reproducible values in PLOP DBs
```

Perception. It is the flip side. I know there is something out in the universe, and I want to see it. I want to perceive it. And the word perception is an interesting word. It really means to take in the entirety of something. And it is very important, because you need to take things in the entirety to get that value proposition from before.

So in the small, again, it is really straightforward. If you are programming with values, if I can reach your value however – you passed it to me, or it is in a collection that I can see. If I can reach it, I can "see" it. I can perceive it. My part of the program can capture that value. Because I know it is never going to change. So as soon as I can reach it, I have acquired it.

Places, it is amazing how difficult this activity is. How can you perceive a mutable object that has more than one getter? What is the way you do it? How do you it? We were all object oriented programmers at one point in time. Who knows how to do this? Who could say right now how to do this?

No one can, right? Yeah I know, Stu is ... He cannot. I know he cannot, from personal experience.

You cannot do this, because you need this other thing. You need the recipe for doing this, and the recipe is something that everybody has to make up over and over and over again. The copying recipe. The locking recipe. The cloning recipe. We have got to make this stuff up. Because the thing could change, and we have multiple independent operational interfaces to the parts. We cannot actually perceive the whole. Cannot do it without help. Without these recipes. And again, we know those recipes do not aggregate.

[Time 0:39:03]

Same thing in the large. On the wire, we do not go and chat with an operational interface to a thing and grab its pieces. Imagine doing that. Imagine if HTTP, in order to get a web page you had to say, "Get the header. Get the cookies. Get the this. Get the that. Get the other thing. Get the title. Get the first segment. Get this div. Get that div. Blah blah blah".

Forget about communication overhead. You could not actually know that the end of all that communication was the page that anybody ever wrote at one point in time. That was something that somebody looked at and said, "Yeah, that is what I intended. That is the value of the page right now." Because the operational interface is in the way of you perceiving the entirety of the thing.

We do not do that. We do not do that on the web. You ask for the page, you get the whole page. You get the entire value.

It is a little bit trickier with databases. Many databases will give you the ability to capture, as a value, some subsection of what they have in a coherent way. But beyond that, they either cannot, or they require a transaction to do that. We will talk about that in a second.

[Time 0:40:13]

```
slide title: Memory

+ In the small
  + Values: remembering == aliasing
  + _Places_: copy, if you can
+ In the large
  + What if there were no permalinks?
  + Place-oriented DBs - DIY time
```

So what about memory? It is very important for our programs, and for the users of our programs, that our programs remember things at various points in time.

So what does it mean to remember something? In the small, again, there is really just nothing to it. Remembering is aliasing. If I can touch the value, I can remember it. I can keep a copy of that indefinitely.

With places, I am really in trouble. I am back to that copy problem. I need to copy it if I want to remember it, because I know its mutable lifetime is going to take it to different values. So I need to be able to copy it, if I can.

In the large, the same thing comes about. How many people remember the early days of the web? At first it was all static pages. It was great. You go to the page, you get the static page. Maybe people updated it, or whatever.

Then people had web sites that were based around programs. They were like, "Oh, cool, I can generate pages. This is awesome!" Who, around that time period of the web when that was first possible, ever said, "Ooh, I am researching this thing. I am going to bookmark all of these things I encountered that

are interesting and relevant to what I am doing"? And then went back to those bookmarks a month later to find that absolutely none of them pointed to the thing that you were looking at before? And actually, you had remembered nothing?

Eventually we figured this out, and have these conventions about permalinks and things like that. But again, if you do not have something like that, you do not actually have a memory system. You only have places out there.

And the same problem happens with the database. If I want to remember something in a database, how am I going to do that? Because people are talking about databases, and databases that lose track of things, and people will say, "We do not. We only add stuff to our database." But you are doing it yourself then. You are doing it yourself. You are saying: I have this place oriented thing. I am not going to use it in that way. I am going to maintain time myself, and I am going to keep new values myself.

And you can do that. How many people have ever written a system that made a new record for every new piece of data, and kept time stamps on those records? How many people wrote the "now" query for that? And was that fun? That is not fun. How many people tried to make that "now" query fast? Not fun. Very, very difficult. So you do not want to do it yourself there.

[Time 0:42:30]

```
slide title: Reduced Coordination

+ In the small
  + Values: No locks!
  + _Places_: Lock policies don't aggregate
+ In the large
  + No read transactions!
  + PLOP: Often gotten wrong
```

Reduced coordination is another critical benefit of values.

In the small when you are programming with values, there simply is not a question about this. It is not a question to answer. It is a question that does not come up. There is no such thing as contention for values.

And the problem we saw about places exists here. The lock policies do not aggregate. We have to lock, and we cannot combine those policies.

In the large, this is another big problem for databases. When you have a place oriented database, if you want to read consistently, you have to read in a transaction. You have to go to the database server and hold up the world, to some degree, in order to see something coherent. It is really a coordination problem. It is an architectural problem. It is a throughput problem. And it is a scaling problem. This is a big architectural disadvantage of place oriented programming. And I think it really highlights one of the big wrongness here.

In addition, even if you think, "All right, I know I have to do read transactions", this is one of the things that is most frequently gotten wrong. People just do not understand read committed, or how read committed is combined, or how independent reads in a batch file work. How many people think the programmers in their shop actually do not know how that stuff works? Yeah. They do not. They really do not.

[Time 0:43:53]

```
slide title: Location Flexibility
```

```
+ In the small
  + Values: aliasing means only one copy
  + _Places_: master copy is special
+ In the large
  + Cache (e.g. HTTP caching)
  + CDN etc
  + Data-based interface is movable
```

Another key benefit of values is location flexibility.

In the small, again, with values there is no need for more than one location, because aliasing covers every case we have seen so far: memory, perception, conveyance. It is all covered by the fact that you only need one copy.

On the flip side, with place oriented programming, there is very specialness to that master copy. If I want to know the value, I have got to manipulate that master copy, and coordinate with everybody about doing that, which means where that is starts to matter to me.

In the large, again, we do this and we really care about it. We have incorporated those things in the HTTP protocol, and whatnot, so that we can do caching. We can say: this expires. This value *is* stable, and therefore you do not need to come to me every time to figure out what it is. You can go to this cache over here. You can go to this content distribution network over there.

One of the interesting things about content distribution networks is: why don't we have CDNs for databases? Why do they make sense for web pages, but not databases? It does not make any sense to me.

The other thing we saw is that database interfaces are movable, inherently. I do not really care where you are. We are going to communicate data. I do not care what language you are implemented in, or if you move around, or if I have to redirect to get to your things like that. So again, I think we understand this in the large, except in the data storage, but definitely in the communication protocols.

[Time 0:45:23]

```
slide title: Facts are Values

+ Not places
+ Don't facts change?
  + _No_ - they incorporate _time_
+ _Fact_ - 'an event or thing known to have happened or existed'
+ From: _factum_ - 'something done'
```

Now the big point. Facts, the things we said are the source of information, are values, by all the definitions I have given, and have all of the benefits we have said. They are not places.

But don't facts change? Didn't my friend get a new email address? Didn't that change the fact of his email address? No, it did not! There is now a new fact, which is: today, your friend's email address is this. It did not change the fact that yesterday, your friend's email address was that. They don't change.

And this goes down to the very core of what fact means. A fact means: something that happened. Something that existed. That is what "fact" means. It does not mean: the slot where you keep your friend's email address. We all laughed at that slide earlier, but it is true. This is what a fact is.

And the roots of the word "fact" actually go all the way back to Latin, where it was a past participle. It said: something done. Factum. Something done. Something that happened.

[Time 0:46:38]

```
slide title: Facts != Recent Facts

+ Knowledge is derived from facts
  + Comparing
  + Combining
+ Especially from different time points
+ You cannot update a fact
  + any more than you can change the past
```

So this is really critical if we want to build information systems, because information is based around facts. And facts does not mean just the most recent facts. We know knowledge is derived from facts. We compare facts to each other. We combine them. We make decisions about that. But one of the critical things we do all of the time is compare facts from different time points.

Imagine if you only knew the present value of every fact that is relevant to you. You only knew the present value of everything. What kind of decision making power would you have, compared to knowing something about time? It would be dramatically reduced. I don't know if anybody saw the film "Memento". If you had a limited window of time over which you knew what had happened, or if you only knew the present, you actually cannot make decisions. It is like built into our brains. If your programs are serving humans, it is built into our brains to delta now with before. That is how we make decisions.

So building systems that only keep the most recent values of things, those are not information systems.

So you cannot update a fact. There is no such thing as updating a fact. A fact is not a place. You cannot do that any more than you can change the past.

[Time 0:47:51]

```
slide title: Information Systems

+ Are fundamentally about _facts_
  + Maintaining, manipulating
+ To give users _leverage_
  + Making decisions
+ Systems should be _value-oriented_
  + Don't use _process_ constructs for information
```

So now let us revisit information systems. What *should* they be? They should be about facts. They should be completely about maintaining and manipulating facts. They should be about giving our users leverage over facts. Helping them make decisions, based upon the facts that the system is maintaining on their behalf.

That means that our systems should be value oriented. They should not be place oriented. We have to stop using process constructs for information. I am not trying to bash objects universally. You can go home and do that. But the very few places where they are appropriate are more process oriented places, and their use for information is actually an idea bereft of merit. There is not one good component of using mutable objects for information. It is just wrong.

[Time 0:48:50]

```
slide title: Decision Making

+ We know what it takes to support our own decision making (hint: _information_)
+ Compare present to past
+ Spot trends, rates
+ Aggregates
+ Often requires _time_
```

We know this is wrong. Because we are decision makers. We all do stuff all of the time. We know what it takes to support our own decision making process. And it is information. We build systems. We run shops. We have stuff we have to accomplish. We are like mini businesses. A programming shop is like a mini business. It has stuff it has to accomplish. It has successes and goals and objectives.

We need to compare the present to the past. We try to spot trends and rates and things, especially in our own systems. We often need to aggregate things. This is what decision making is about, both in the large and in the small, and for businesses and for programmers.

And a lot of this decision making requires a time component.

[Time 0:49:34]

```
slide title: Programmer I.T.

+ Source Control
  + Update in place? - No
  + Timestamps - of course!
+ Logs
  + Update in place? - No
  + Timestamps - of course!
```

So let us look at programmer I.T., because we also use computers to support our own decision making process, don't we? What kinds of systems do we give ourselves? What kind of information systems do we give ourselves?

Well one big one is source control. Anybody keep their source control in a directory, where when somebody edits the file, they save it into the directory over the one that is there? Update in place? How many people do update in place source control? No! We don't do that.

How many people store the stuff in source control with no dates on the edits? No timestamps? No! We keep track of time.

How many people throw away their old source code? No! We don't do that. Why would we do that? We could not make decisions about what was happening in our business, in our programming business. We would be crippled by that.

What is another critical thing that we keep track of? We run our programs, and we keep track of what our programs do *in logs*. Because we need to look at those logs. We need to make decisions about: is our program working? Is it working well? Is it using memory well? Does it have good performance? If there was a problem, what went wrong? All of this decision making we need to do as our little programming business.

Are logs update in place? Anybody want to use a log system that only keeps track of the last latency, or the last time somebody communicated with a particular endpoint? Anybody want to do that? No! No, they are not update in place.

Anybody have a log that has no timestamps in it? No! We keep timestamps. Of course we do. How are we supposed to make decisions without timestamps, and without keeping track of everything that happened? Because we want the facts when we need to make decisions. This is our I.T.

Anybody want to have the ability for somebody to go back to an old version of a source file and change the old version in place? Or change logs in place? Edit the logs? Any value in that? No. Our I.T. systems are not like this.

[Time 0:51:44]

```
slide title: Big Data

+ Business to programmers:
  + _"I like your database better than the one you gave me"_
+ Logs have _all_ the information
  + and timestamps
+ We are reactive here
  + mining logs, seriously?
+ Not delivering leverage
```

So let us talk about big data. It is my contention that a certain portion, and quite possibly a very large portion, of big data, the hot new topic, the big thing, is *this*. It is businesses saying to programmers, "I like your database better than the one you gave me, because your database has everything in it. The one that you gave me, it only remembers the last thing". I cannot track trends. I cannot see latencies. I cannot see where everybody was on the site. I cannot make the same kinds of decisions that you can. I want to mine your logs to get out business critical information, because that is the only place that it exists. Because you are not keeping your own database this way.

These logs have everything. They have time on them. They are huge, rich sources of decision making power. They are all filled with stuff we are not putting in the database, for some reason.

And I think it is actually quite embarrassing. I think that I.T. right now is in a very reactive place here. Business has discovered the value that was in our logs. Our logs were just for ourselves, to see if our programs were working. But they happen to keep track of where everybody clicked, how long things took, what the flow was between events, and everything that happened, including stuff that happened that in the database ended up overwriting old stuff. That is all in these logs.

[Time 0:53:06]

But really, mining logs? Flat files? We know better than that. We have technology that is better than flat files. Is anybody really happy that their logs are in flat files in the end? Obviously it is efficient to sort of append on them, but in the end, you struggle after that to try to get leverage over that data. Because we know flat files are not great. One of the advantages programmers gave to businesses was the invention of databases, the invention of indexes and trees, and these other data structures that really let people leverage information. We are not actually putting this critical information into a leveragable place right now.

And it is co-mingled with a bunch of crap that is not actually useful to the business, like latencies and things that were really to communicate with us. There is this mix in the logs of stuff about seeing if the system is working OK, and actual activity against the business. If you could pull out this part [useful to the business], and put it in a leveragable store, your businesses would be a lot happier.

And that is where we are going to end up. Big data is forcing us to do that. But you should all look at the deep reasons why this is happening. They have to do with the fact that we have built better information systems for ourselves than we have delivered to our business customers.

[Time 0:54:26]

```
slide title: The Space Age

+ _Space_
  + 'The unlimited expanse in which all things are located, and all events occur'
+ If _new_ never fails ...
  + you are effectively running in space
+ If _S3_ never fills up ...
  + it is not the _cloud_, but space
```

So I think we are entering the space age. And the space age is the age where we have access to space from our programs. We said place was a portion of space. Space is the unlimited expanse in which all things exist and all events occur.

And this definition of space, it goes all the way back to the roots for "spatial" in the Latin there, it has *always* incorporated both place and time. The notion of space has always encompassed both place and time. They are connected together, quite significantly.

Are we in the space age? Are we programming now with space? I think we are. We had virtual memory, which really took us a level away from the actual addresses. Then we had GC, which meant more was sort of transparently available. Whenever I needed more, I could get it.

Guess what? If your program runs indefinitely long, and calls new, and new never fails, your program is running in space, not in a place. It is running in space.

If S3 never fills up, or if you can always go to the store and buy another hard drive and stick it in your array, live, you are programming with space. There is no place there. There is no limit. There is no delimiting that. That is space.

What does this mean? It means that we can take a different approach to the way we do things. We are going to say: we *are* building information systems. Those systems should be maintaining facts.

[Time 0:56:08]

```
slide title: New Facts, New Space

+ The end of PLOP
+ If you can afford this, why do anything else?
+ You can afford this
  + _(there will be garbage)_
```

And that new facts require new space, and we have space. This is the end, or this should call for an end, to place oriented programming. If you can afford to do this, why would you do anything else? What is a really good reason for doing something else?

And guess what? You can afford to do this. You already can afford it. You are already running programs that call new indefinitely, and do not fail. And you have access to S3, or things like it.

There will be garbage. There will be different characteristics to our use of space, especially storage, that are very analogous to what we saw when we enabled space in memory. The whole notion of garbage collection is going to happen in storage. But if you have a grip on that, you have no problems understanding a space orientation with storage.

[Time 0:57:06]

```
slide title: Summary
```

```
+ We continue to use place-oriented programming languages and databases
  + _and make new ones!_
  + long after rationale is _gone_
+ We are missing out on the value of values
  + which we recognize
+ We need to deliver _information_ systems
  + demand is clear, resources available
```

So to summarize, for some reason we continue to use place-oriented programming, both languages and databases. We even make new ones. It is actually the saddest thing, is the fact that we continue to make *new* languages and *new* databases that still emulate the decisions that were made when computers were tiny, and we needed to program in a place as opposed to in space.

This is not a NoSQL versus old SQL. There are older systems that maintain time correctly, and most NoSQL and new SQL things are still place oriented. It is not about old and new. It is about place oriented or not. The rationale for doing this is gone.

I do not think anyone could deny the benefits that I just enumerated for values, and in fact by your actions, especially in your own I.T. services for yourself, you are demonstrating you know this value proposition. We have recognized this.

So we need to start making information systems that are really about facts. That are really about information. I think the demand is clear. This big data explosion is saying, businesses are saying, I demand to know everything that happened. I demand not to lose track of the facts. These things are important to me. They are important to the decisions my business makes. And they are soon going to say: Why the hell does that only exist in a log? Why are you sticking it there? You have indexes in databases and things like that you can use for this.

[Time 0:58:47]

```
slide:

Facts do not cease to exist because they are ignored.

                Aldous Huxley
```

I will leave you with this, and thanks very much.

[Audience applause]

[Time 0:58:50]