Simple Made Easy

- **Speaker: Rich Hickey**
- **Conference: Strange Loop 2011 - Sept 2011**
- **Video: http://www.infoq.com/presentations/Simple-Made-Easy**

[Time 0:00:00]

```
slide:

Simple Made Easy

Rich Hickey
```

Hi. So who is ready for some more category theory?

[Audience applause]

You are all in the wrong room.

[Audience laughter]

This talk I hope seems deceptively obvious. One of the things that is great about this conference is this is a pretty cutting edge crowd. A lot of you are adopting new technologies. A lot of you are doing functional programming. And you may be nodding, saying yeah, yeah, yeah through parts of this. And if some of it is familiar, that is great. On the other hand, I think that I would hope that you would come away from this talk with some tools you could use to help conduct a similar kind of discussion to this talk with other people that you are trying to convince to do the right thing.

[Time 0:01:00]

```
slide:

Simplicity is prerequisite for reliability

            Edsger W. Dijkstra
```

So, I will start with an appeal to authority. Simplicity is a prerequisite for reliability. I certainly agree with this. I do not agree with everything Dijkstra said, and I think he might have been very wrong about proof in particular. But I think he is right about this. We need to build simple systems if we want to build good systems. I do not think we focus enough on that.

[Time 0:01:23]

```
slide title: Word Origins

+ Simple              + Easy

  sim- plex             ease < aise < adjacens

  one fold/braid        lie near

  vs _complex_          vs _hard_
```

I love word origins. They are tremendous fun. One of the reasons why they are fun is because words eventually come to mean whatever we all accept them to mean. Whatever is commonly understood to be the meaning is what it means. And it is often interesting to say, well, I wish I could; I wish we could go back to what it really

means and use that. And I think there is a couple of words that I am going to use in this talk that I would love for you to come away knowing the origins of and try to use more precisely, especially when talking about software.

So the first word is "simple". And the roots of this word are "sim" and "plex", and that means one fold or one braid or twist. And that characteristic about being about one - literally - fold or twist . . . Of course one twist, what is one twist look like? No twists, actually.

And the opposite of this word is "complex", which means braided together or folded together. Being able to think about our software in terms of whether or not it is folded together is sort of the central point of this talk.

The other word we frequently use interchangeably with "simple" is the word "easy". And the derivation there is to a French word, and the last step of this derivation is actually speculative, but I bought it because it serves this talk really well, and that is from the Latin word that is the root of "adjacent", which means "to lie near" and "to be nearby". And the opposite is "hard". Of course, the root of "hard" has nothing to do with lying near. It does not mean "lie far away". It actually means "strong", or torturously so.

[Time 0:03:17]

```
slide title: Simple

+ One fold/braid      + But not
  + One role            + One instance
  + One task            + One operation
  + One concept       + _About lack of
  + One dimension       interleaving, not
                        cardinality_
                      + _Objective_
```

So if we want to try to apply "simple" to the kinds of work that we do, we are going to start with this concept of having one braid. And look at it in a few different dimensions. I thought it was interesting in Eric's talk to talk about dimensions, because it is definitely a big part of doing design work. And so if we want to look for simple things, we want to look for things that have sort of *one* of something. They have one role. They fulfill one task or job. They are about accomplishing sort of one objective. They might be about one concept, like security.

And sort of overlapping with that is they may be about a particular dimension of the problem that you are trying to solve. The critical thing there, though, is that when you are looking for something that is simple, you want to see it have focus in these areas. You do not want to see it combining things.

On the other hand, we cannot get too fixated about "one". In particular, simple does not mean that there is only one of them. It also does not mean an interface that only has one operation. So it is important to distinguish cardinality, counting things, from actual interleaving. What matters for simplicity is that there is no interleaving, not that there is only one thing. And that is very important.

The other critical thing about simple, as we have just described it, is if something is interleaved or not, that is sort of an objective thing. You can probably go and look and see. I do not see any connections. I do not see anywhere where this twists with something else. So simple is actually an objective notion. That is also very important in deciding the difference between simple and easy.

[Time 0:05:16]

```
slide title: Easy

+ Near, at hand              + Near our capabilities
  + on our hard drive, in    + Easy is _relative_
```

```
    our tool set, IDE, apt
    get, gem install ...
  + Near to our
    understanding/skill set
    + familiar
```

So let us look at "easy". I think this notion of nearness is really, really cool. In particular, obviously there are many ways in which something can be near. There is sort of the physical notion of being near. Is something right there? And I think that is where the root of the word came from. This is easy to obtain because it is nearby. It is not in the next town. I do not have to take a horse or whatever to go get to it.

We do not have the same notion of physicality necessarily in our software, but we do sort of have our own hard drive or our own toolset, or it is sort of the ability to make things physically near by getting them through things like installers and stuff like that.

The second notion of nearness is something being near to our understanding, or in our current skill set. And I do not mean in this case near to our understanding meaning a capability. I mean literally near something that we already know. So the word in this case is about being familiar.

I think that, collectively, we are infatuated with these two notions of easy. We are just so self-involved in these two aspects; it is hurting us tremendously. All we care about is: can I get this instantly and start running it in five seconds? It could be this giant hairball that you got, but all you care is: can you get it?

In addition, we are fixated on, "oh, I cannot read that". Now I cannot read German. Does that mean German is unreadable? No. I do not know German. So this sort of approach is definitely not helpful. In particular, if you want everything to be familiar, you will never learn anything new because it cannot be significantly different from what you already know and not drift away from the familiarity.

[Time 0:07:13]

There is a third aspect of being easy that I do not think we think enough about that is going to become critical to this discussion, which now is being near to our capabilities. And we do not like to talk about this because it makes us uncomfortable, because what kind of capabilities are we talking about? If we are talking about easy in the case of violin playing or piano playing or mountain climbing or something like that, well, I do not personally feel bad if I do not play the violin well because I do not play the violin at all.

But the work that we are in is conceptual work, so when we start talking about something being outside of our capability, it really starts trampling on our egos in a big way. And so due to a combination of hubris and insecurity, we never really talk about whether or not something is outside of our capabilities. It ends up that it is not so embarrassing after all, because we do not have tremendously divergent abilities in that area.

The last thing I want to say about "easy", and the critical thing to distinguish it from "simple", is that easy is relative. Playing the violin and reading German are really hard for me. They are easy for other people, certain other people. So unlike simple where we can go and look for interleavings, look for braiding, easy is always going to be: easy for whom, or hard for whom? It is a relative term.

The fact that we throw these things around sort of casually saying, "Oh, I like to use that technology because it is simple," and when I am saying simple, I mean easy. And when I am saying easy, I mean because I already know something that looks very much alike that. It is how this whole thing degrades and we can never have an objective discussion about the qualities that matter to us in our software.

[Time 0:09:05]

```
slide title: Construct vs Artifact

+ We focus on experience of use of construct
  + programmer convenience
```

```
  + programmer replaceability
+ Rather than the long term results of use
  + software quality, correctness
  + maintenance, change
+ _We must assess constructs by their artifacts_
```

So, what is one critical area where we have to distinguish these two things, and look at them from a perspective of them being easy and being simple? It has to do with constructs and artifacts. We program with constructs. We have programming languages. We use particular libraries. And those things, in and of themselves, when we look at them, like when we look at the code we write, have certain characteristics in and of themselves.

But we are in a business of artifacts. We do not ship source code, and the user does not look at our source code and say, "Ah, that is so pleasant." Right? No? They run our software, and they run it for a long period of time. And, over time, we keep glomming more stuff on our software. All of that stuff, the running of it, the performance of it, the ability to change it, all is an attribute of the artifact, not the original construct.

But again, here we still focus so much on our experience of the use of the construct. Oh, look; I only had to type 16 characters. Wow! That is great. No semicolons or things like that. This whole notion of sort of programmer convenience, again, we are *infatuated* with it, not to our benefit.

[Time 0:10:29]

On the flipside it gets even worse. Our employers are also infatuated with it. Those first two meanings of easy, what do they mean? If I can get another programmer in here, and they look at your source code, and they think it is familiar, and they already know the toolkit? So it is near at hand. They have always had the same tool in their toolkit. They can read it. I can replace you. It is a breeze, especially if I ignore the third notion of easy, which is whether or not anybody can understand your code. Because they do not actually care about that. They just care that somebody can go sit in your seat, start typing.

So again, as sort of business owners, there is sort of, again, the same kind of focus on those first two aspects of easy because it makes programmers replaceable.

So we are going to contrast this with the impacts of long-term use. What does it mean to use this long term? And what is there? What is there is all the meat. Does the software do what it is supposed to do? Is it of high quality? Can we rely on it doing what it is supposed to do? Can we fix problems when they arise? And if we are given a new requirement, can we change it?

These things have nothing to do with the construct, as we typed it in, or very little to do with it, and have a lot to do with the attributes of the artifact. We have to start assessing our constructs based around the artifacts, not around the look and feel of the experience of typing it in or the cultural aspects of that.

[Time 0:12:13]

```
slide title: Limits

+ We can only hope to make reliable those things we can understand
+ We can only consider a few things at a time
+ Intertwined things must be considered together
+ Complexity undermines understanding

[ Drawing showing a person stick figure juggling three balls. ]
```

So let us talk a little bit about limits. Oh, look; it does move. This is just supposed to sort of lull you into this state where everything I say seems true.

[Audience laughter]

Because I cannot use monads to do that.

[Audience laughter]

This stuff is pretty simple logic. How can we possibly make things that are reliable that we do not understand? It is very, very difficult. I think Professor Sussman made a great point saying, "there is going to be this tradeoff". As we make things more flexible and extensible and dynamic in some possible futures for some kinds of systems, we are going to make a tradeoff in our ability to understand their behavior and make sure that they are correct. But for the things that we want to understand and make sure are correct, we are going to be limited to our understanding.

And our understanding is very limited. There is the whole notion of how many balls can you keep in the air at the time, or how many things can you keep in mind at a time? It is a limited number, and it is a very small number. So we can only consider a few things and, when things are intertwined together, we lose the ability to take them in isolation.

So if every time I think I pull out a new part of the software I need to comprehend, and it is attached to another thing, I have to pull that other thing into my mind because I cannot think about the one without the other. That is the nature of them being intertwined. So every intertwining is adding this burden, and the burden is kind of combinatorial as to the number of things that we can consider. So, fundamentally, this complexity, and by complexity I mean this braiding together of things, is going to limit our ability to understand our systems.

[Time 0:14:06]

```
slide title: Change

+ Changes to software require analysis and decisions
+ What will be impacted?
+ Where do changes need to be made?
+ Your ability to reason about your program is critical to
  changing it without fear
  + Not talking about proof, just informal reasoning
```

So how do we change our software? Apparently, I heard in a talk today, that Agile and Extreme Programming have shown that refactoring and tests allow us to make change with zero impact.

[Audience laughter]

I never knew that. I still do not know that.

[Audience laughter]

That is not actually a knowable thing. That is phooey.

[Audience laughter]

If you are going to change software, you are going to need to analyze what it does and make decisions about what it ought to do. At least you are going to have to go and say, "What is the impact of this potential change? And what parts of the software do I need to go to to effect the change?"

I do not care if you are using XP or Agile or anything else. You are not going to get around the fact that if you cannot reason about your program, you cannot make these decisions. But I do want to make clear here because a lot of people, as soon as they hear the words reason about, they are like, "Oh, my God! Are you saying that you have to be able to prove programs?" I am not. I do not believe in that. I do not think that is an objective. I am just talking about informal reasoning, the same kind of reasoning we use every day to decide what we are going to do. We do not take out category theory and say, "Woooo," you know. We actually can reason without it. Thank goodness.

[Time 0:15:31]

```
slide title: Debugging

+ What's true of every bug in the field?
+ It has passed the type checker
  + and all the tests
+ Your ability to reason about your program is critical to debugging

[ Photos of guard rails next to automobile roads. ]
```

So what about the other side? There are two things you do with the future of your software. One is, you add new capabilities. The other thing is you fix the ones you did not get done so well.

And I like to ask this question: what is true of every bug found in the field?

[Audience reply: Someone wrote it?][Audience reply: It got written.]

It got written. Yes. What is a more interesting fact about it? It passed the type checker.

[Audience laughter]

What else did it do?

[Audience reply: (Indiscernible)]

It passed all the tests. OK. So now what do you do? I think we are in this world I would like to call guardrail programming. It is really sad. We are like: I can make change because I have tests. Who does that? Who drives their car around banging against the guardrail saying, "Whoa! I am glad I have got these guardrails because I would never make it to the show on time."

[Audience laughter]

Right? And do the guardrails help you get to where you want to go? Do guardrails guide you places? No. There are guardrails everywhere. They do not point your car in any particular direction.

So again, we are going to need to be able to think about our program. It is going to be critical. All of our guardrails will have failed us. We are going to have this problem. We are going to need to be able to reason about our program. Say, "Well, you know what? I think," because maybe if it is not too complex, I will be able to say, "I know, through ordinary logic, it could not be in this part of the program. It must be in that part, and let me go look there first." Things like that.

[Time 0:17:13]

```
slide title: Development Speed

+ Emphasizing ease gives early speed
+ Ignoring complexity will slow you down over the long haul
+ On throwaway or trivial projects, nothing much matters

[ Chart with Time increasing to the right on the X axis, Speed,
meaning development speed increasing on the Y axis.  Made-up chart
labeled "Easy" starts out with high development speed early on, then
gets lower as time progresses.  Made-up chart labeled "Simple" starts
out low or medium, and gradually grows over time. ]
```

Now, of course, everybody is going to start moaning, "But I have all this speed. I am agile. I am fast. This easy stuff is making my life good because I have a lot of speed."

What kind of runner can run as fast as they possibly can from the very start of a race?

[Audience reply: Sprinter]

Only somebody who runs really short races, OK?

[Audience laughter]

But of course, we are programmers, and we are smarter than runners, apparently, because we know how to fix that problem. We just fire the starting pistol every hundred yards and call it a new sprint.

[Audience laughter and applause]

I do not know why they have not figured that out.

It is my contention, based on experience, that if you ignore complexity, you will slow down. You will invariably slow down over the long haul.

Of course, if you are doing something that is really short term, you do not need any of this. You could write it in ones and zeros. And this is my really scientific graph. You notice how none of the axes are – there is no numbers on it because I just completely made it up.

[Audience laughter]

It is an experiential graph, and what it shows is: if you focus on ease and ignore simplicity, so I am not saying you cannot try to do both. That would be great. But if you focus on ease, you will be able to go as fast as possible from the beginning of the race. But no matter what technology you use, or sprints or firing pistols, or whatever, the complexity will eventually kill you. It will kill you in a way that will make every sprint accomplish less. Most sprints be about completely redoing things you have already done. And the net effect is you are not moving forward in any significant way.

Now if you start by focusing on simplicity, why can't you go as fast as possible right at the beginning? Because some tools that are simple are actually as easy to use as some tools that are not. Why cannot you go as fast then?

[Audience response: You have to think.]

You have to think. You have to actually apply some simplicity work to the problem before you start, and that is going to give you this ramp up.

[Time 0:19:35]

```
slide title: Easy Yet Complex?

+ Many complicating constructs are
  + Succinctly described
  + Familiar
  + Available
  + Easy to use
+ What matters is the complexity they _yield_
  + Any such complexity is _incidental_

[ Photo of a person using a loom with yarn. ]
```

So one of the problems I think we have is this conundrum that some things that are easy actually are complex. So let us look.

There are a bunch of constructs that have complex artifacts that are very succinctly described. Some of the things that are really dangerous to use are so simple to describe. They are incredibly familiar. If you are

coming from object-orientation, you are familiar with a lot of complex things. They are very much available. And they are easy to use. In fact, by all measures, conventional measures, you would look at them and say, "This is easy."

But we do not care about that. Again, the user is not looking at our software, and they do not actually care very much about how good a time we had when we were writing it. What they care about is what the program does, and if it works well, it will be related to whether or not the output of those constructs were simple. In other words, what complexity did they yield?

When there is complexity there, we are going to call that incidental complexity. It was not part of what the user asked us to do. We chose a tool. It had some inherent complexity in it. It is incidental to the problem. I did not put the definition in here, but incidental is Latin for "your fault".

[Audience laughter]

And it is. And I think you really have to ask yourself: are you programming with a loom? You are having a great time. You are throwing that shuttle back and forth. And what is coming out the other side is this knotted mess. It may look pretty, but you have this problem. What is the problem? The problem is the knitted castle problem. Do you want a knitted castle?

[Time 0:21:31]

```
slide title: Benefits of Simplicity

+ Ease understanding
+ Ease of change
+ Easier debugging
+ Flexibility
  + policy
  + location etc

[Photo of knitted castle, and another made of Lego blocks. ]
```

What benefits do we get from simplicity? We get ease of understanding. That is sort of definitional. I contend we get ease of change and easier debugging. Other benefits that come out of it, that are sort of on a secondary level, are increased flexibility. And when we talk more about modularity and breaking things apart, we will see where that falls. Like the ability to change policies or move things around. As we make things simpler, we get more independence of decisions because they are not interleaved, so I can make a location decision. It is orthogonal from a performance decision.

And I really do want to ask the question, agile-ist or whatever: Is having a test suite and refactoring tools going to make changing the knitted castle faster than changing the Lego castle? No way. Completely unrelated.

[Time 0:22:34]

```
slide title: Making Things Easy

+ Bring to hand by installing
  + getting approved for use
+ Become familiar by learning, trying
+ But mental capability?
  + not going to move very far
  + make things near by simplifying them
```

OK. So how do we make things easy? Presumably the objective here is not to just bemoan the software crisis. So what can we do to make things easy? So we will look at those aspects of being easy again.

There is a location aspect. Making something at hand, putting it in our toolkit. That is relatively simple. We just install it. Maybe it is a little bit harder because we have to get somebody to say it is OK to use it.

Then there is the aspect of: how do I make it familiar? I may not have ever seen this before. That is a learning exercise. I have got to go get a book, go take a tutorial, have somebody explain it to me. Maybe try it out. Both these things we are driving. We are driving. We install. We learn. It is totally in our hands.

Then we have this other part though, which is the mental capability part. And that is the part that is always hard to talk about, the mental capability part. Because, the fact is, we can learn more things. We actually cannot get much smarter. We are not going to move our brain closer to the complexity. We have to make things near by simplifying them.

But the truth here is not that they are these super, bright people who can do these amazing things and everybody else is stuck, because the juggling analogy is pretty close. The average juggler can do three balls. The most amazing juggler in the world can do 9 balls or 12 or something like that. They cannot do 20 or 100. We are all very limited. Compared to the complexity we can create, we are all statistically at the same point in our ability to understand it, which is not very good. So we are going to have to bring things towards us.

And because we can only juggle so many balls, you have to make a decision. How many of those balls do you want to be incidental complexity and how many do you want to be problem complexity? How many extra balls? Do you want to have somebody throwing you balls that you have to try to incorporate in here? Oh, use this tool. And you are like, whoa! You know, more stuff. Who wants to do that?

[Time 0:24:47]

```
slide title: Parens are Hard!

+ Not at hand for most
+ Nor familiar
+ But are they simple?
+ Not in CL/Scheme
  + overloaded for calls
    and grouping
  + for those that bothered
    trying, this is a valid
    complexity complaint
+ Adding a data structure
  for grouping, e.g.
  vectors, makes each
  simpler
  + minimal effort can then
    make them easy too
```

All right, so let us look at a fact.

[Audience laughter]

I have been on the other side of this complaint, and I like it. We can look at it really quickly only because it is not – this analysis has nothing to do with the usage. This complexity analysis is just about the programmer experience. So parens are hard. They are not at hand for most people who have not otherwise used it.

And what does that mean? It means that they do not have an editor that knows how to do paren matching or move stuff around structurally, or they have one and they have never loaded the mode that makes that happen. Totally given. It is not at hand.

Nor is it familiar. I mean, everybody has seen parentheses, but they have not seen them on that side of the method.

[Audience laughter]

I mean [laughter] that is just crazy!

[Audience laughter]

But I think this is your responsibility to fix these two things, as a user, as a potential user. You have got to do this.

[Time 0:25:58]

But we could dig deeper. Let us look at the third thing. Did you actually give me something that was simple? Is a language built all out of parens simple? In the case I am saying, is it free of interleaving and braiding? And the answer is no.

Common Lisp and Scheme are not simple in this sense, in their use of parens, because the use of parentheses in those languages is overloaded. Parens wrap calls. They wrap grouping. They wrap data structures. And that overloading is a form of complexity by the definition I gave you.

And so, if you actually bothered to get your editor set up and learn that the parenthesis goes on the other side of the verb, this was still a valid complaint. Now, of course, everybody was saying easy, and it is hard, it is complex, and they were using these words really weakly. But it was hard for a couple reasons you could solve, and it was not simple for a reason that was the fault of the language designer, which was that there was overloading there. And we can fix that. We can just add another data structure.

It does not make Lisp not Lisp to have more data structures. It is still a language defined in terms of its own data structures. But having more data structures in play means that we can get rid of this overloading in this case, which then makes it your fault again, because now the simplicity is back in the construct, and it is just a familiarity thing, which you can solve for yourself.

[Time 0:27:33]

```
slide:

LISP programmers know the value of everything and the cost of nothing.

                          Alan Perlis
```

OK. This was an old dig at Lisp programmers. I am not totally sure what he was talking about. I believe it was a performance related thing. That Lispers, they consed up all this memory, and they did all this evaluation, and it was a pig. Lisp programs at that time were complete pigs relative to the hardware. So they knew the value of all these constructs, this dynamic nature. These things are all great. They are valuable, but there was this performance cost.

I would like to lift this whole phrase and apply it to all of us right now. As programmers, we are looking at all kinds of things, and I just see it. Read Hacker News or whatever. It is like, oh, look; this thing has this benefit. Oh, great. I am going to do that. Oh, but this has this benefit. Oh, that is cool. Oh, that is awesome. You know, that is shorter. You never see in these discussions: was there a tradeoff? Is there any downside? Is there anything bad that comes along with this? Never. Nothing.

It is just like we look all for benefits. So as programmers now, I think we are looking all for benefits, and we are not looking carefully enough at the byproducts.

[Time 0:28:49]

```
slide title: What's in _your_ Toolkit?

     Complexity            Simplicity
```

```
--------------------    ----------------------------

  State, Objects              Values

       Methods             Functions, Namespaces

        vars                Managed refs

Inheritance, switch,      Polymorphism a la carte
     matching

       Syntax                   Data

Imperative loops, fold     Set functions

       Actors                  Queues

        ORM                Declarative data manipulation

   Conditionals                Rules

  Inconsistency              Consistency
```

So, what is in your toolkit? I have these two columns. One says complexity and one says simplicity. The simplicity column just means simpler. It does not mean that the things over there are purely simple. Now I did not label these things bad and good. I am leaving your minds to do that.

[Audience laughter]

So what things are complex and what are the simple replacements? I am going to dig into the details on these, so I will not actually explain why they are complex, but I am going to state and objects are complex, and values are simple and can replace them in many cases. I am going to say methods are complex, and functions are simple. And namespaces are simple. The reason why methods are there are because often the space of methods, the class or whatever, is also a mini, very poor namespace.

Vars are complex and variables are complex. Managed references are also complex, but they are simpler. Inheritance, switch statements, pattern matching are all complex, and polymorphism a la carte is simple.

[Time 0:30:11]

Now remember the meaning of simple. The meaning of simple means unentangled, not twisted together with something else. It does not mean I already know what it means. Simple does not mean, "I already know what it means".

OK. Syntax is complex. Data is simple. Imperative loops, fold even, which seems kind of higher level, still has some implications that tie two things together, whereas set functions are simpler. Actors are complex, and queues are simpler. ORM is complex, and declarative data manipulation is simpler. Even Eric said that in his talk. He said it really fast near the end.

[Audience laughter]

Oh, yeah, and eventual consistency is really hard for programmers.

Conditionals are complex in interesting ways, and rules can be simpler. And inconsistency is very complex. It is almost definitionally complex, because consistent means to stand together, so inconsistent means to stand apart. That means taking a set of things that are standing apart and trying to think about them all at the

same time. It is inherently complex to do that. And anybody who has tried to use a system that is eventually consistent knows that.

[Time 0:31:35]

```
slide title: Complect

+ _To interleave, entwine, braid_
  + archaic
+ Don't do it!
  + Complecting things is the source of complexity
+ Best to avoid in the first place

[ Drawing shows 6 steps, starting from having four parallel strands,
  and step by step braiding them together. ]
```

OK. So there is this really cool word called "complect". I found it.

[Audience laughter]

I love it. It means "to interleave or entwine or braid". OK? I want to start talking about what we do to our software that makes it bad. And I do not want to say braid or entwine because it does not really have the good/bad connotation that complect has. Complect is obviously bad.

[Audience laughter]

It happens to be an archaic word, but there are no rules that say you cannot start using them again, so I am going to use them for the rest of the talk.

So what do you know about complect? It is bad. Do not do it. This is where complexity comes from: complecting. It is very simple.

[Audience laughter]

And in particular, it is something that you want to avoid in the first place. Look at this diagram. Look at the first one. Look at the last one. It is the same stuff in both those diagrams. It is the same strips! What happened? They got complected.

[Audience laughter]

And now it is hard to understand the bottom diagram from the top one, but it is the same stuff. You are doing this all the time. You can make a program a hundred different ways. Some of them, it is just hanging there. It is all straight. You look at it. You say, I see it is four lines, this program. Then you could type in four lines in another language or with a different construct, and you end up with this knot, so we have got to take care of that.

[Time 0:33:07]

```
slide title: Compose

+ _To place together_
+ Composing simple components is the key to robust software

[ Drawing of two Lego blocks apart, then connected together. ]
```

So "complect" actually means to braid together. And "compose" means to place together. And we know that. Everybody keeps telling us. What we want to do is make composable systems. We just want to place things

together, which is great, and I think there is no disagreement. Composing simple components, simple in that same respect, is the way we write robust software.

[Time 0:33:34]

```
slide title: Modularity and Simplicity

[ Blue Lego block on the left, yellow Lego block on the right. ]
```

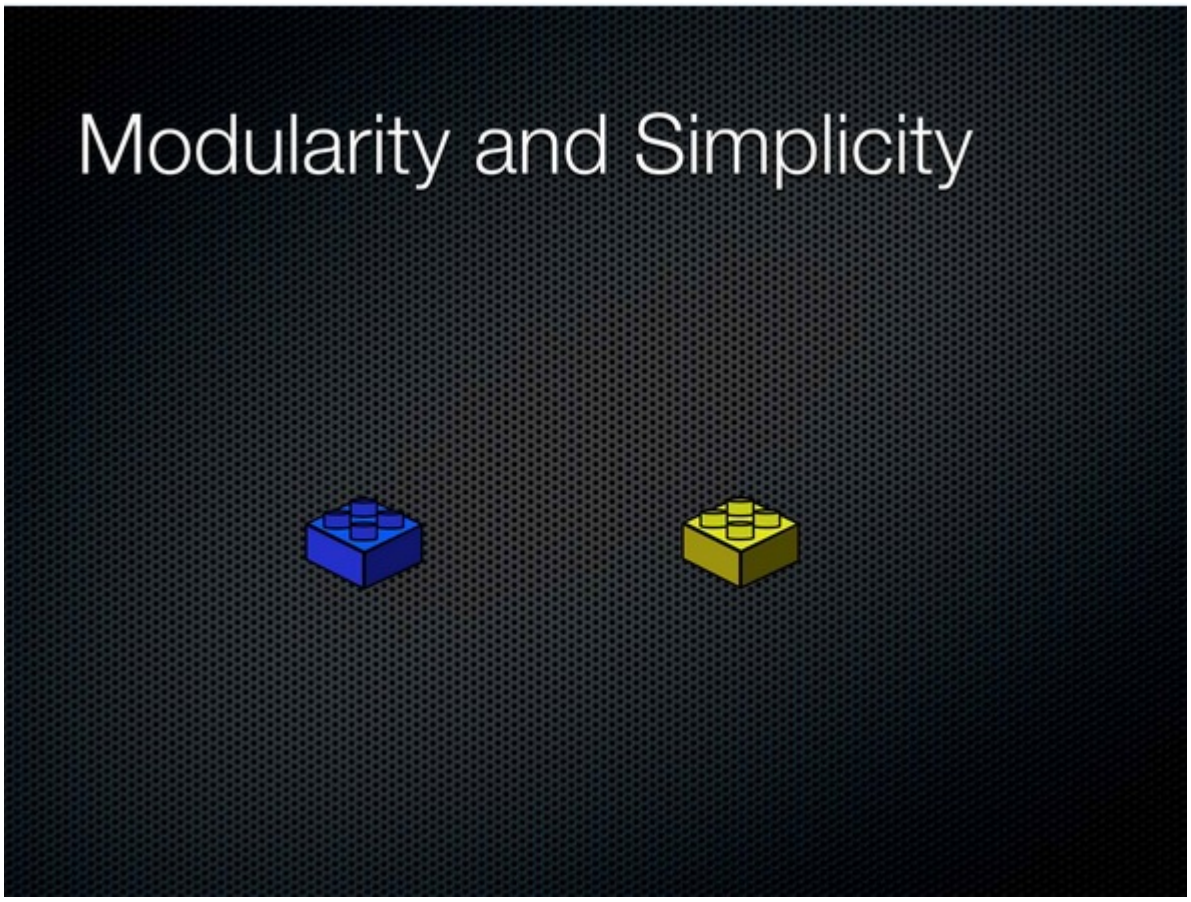

Figure 1: 00:33:35 Modularity and Simplicity

So it is simple. All we need to do is – everybody knows this. I am up here just telling you stuff you know. We can make simple systems by making them modular. We are done. I am halfway through my talk. I do not even know if I am going to finish. It is so simple. This is it. This is the key.

No, it is obviously not the key. Who has seen components that have this kind of characteristic? I will raise my hand twice because not enough people are raising their hands. It is ridiculous, right? What happens? You can write modular software with all kinds of interconnections between them. They may not call each other, but they are completely complected.

```
slide title: Modularity and Simplicity

[ Blue Lego block on the left now has "thought balloon" showing that
it is thinking about the yellow Lego block, and the yellow Lego block
on the right has similar thought balloon showing it thinking about the
blue Lego block. ]
```

And we know how to solve this. It has nothing to do with the fact that there are two things. It has to do with what those two things are allowed to think about, if you want to really anthropomorphize.

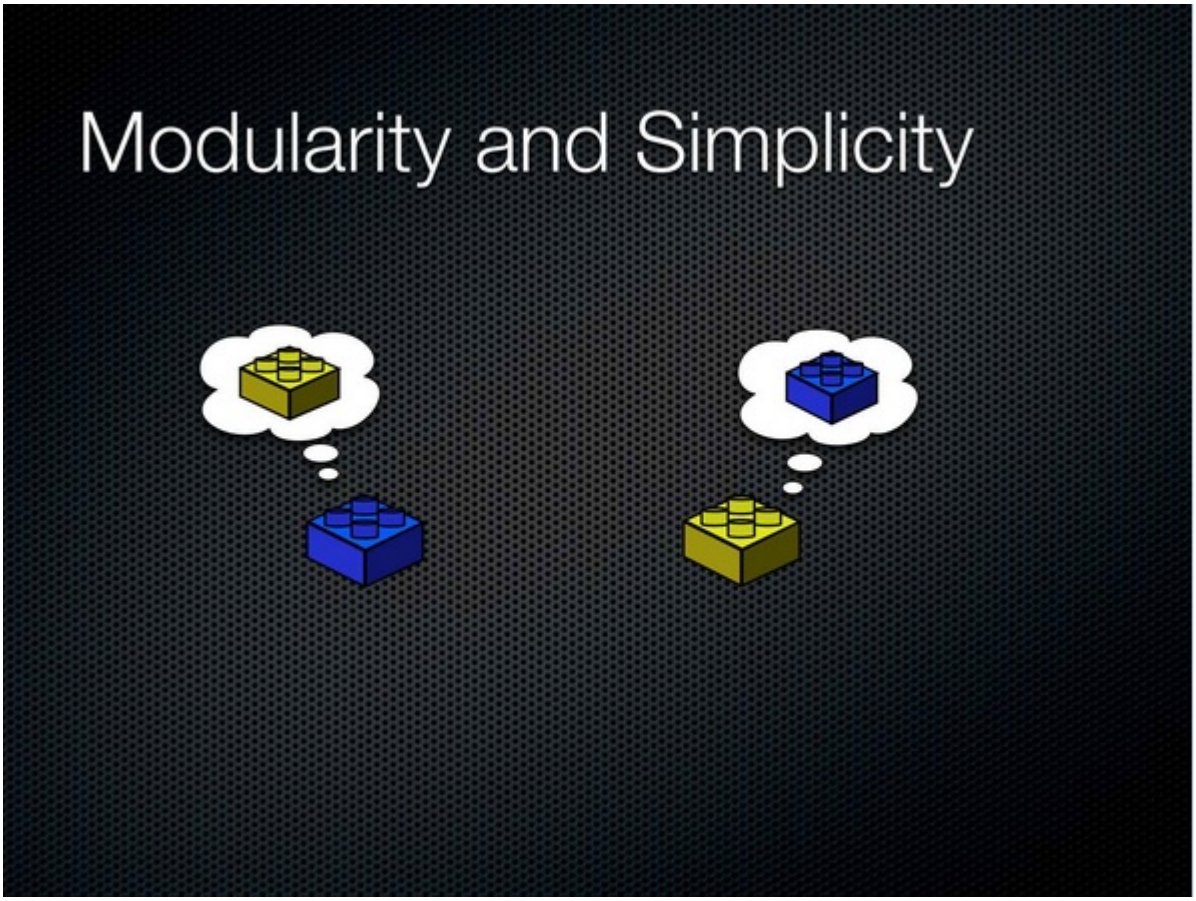

Figure 2: 00:34:19 Modularity and Simplicity - build slide

[Time 0:34:31]

```
slide title: Modularity and Simplicity

[ Blue Lego block on the left now has "thought balloon" showing that
it is thinking about just the white surface of the yellow Lego block,
and the yellow Lego block on the right has similar thought balloon
showing it thinking about only the white surface of the blue Lego
block. ]

+ _Partitioning and stratification don't imply simplicity_
  + but _are_ enabled by it
+ Don't be fooled by code organization
```

And what do we want to make things allowed to think about, and only these things? Some abstractions. I do not know if that is coming out that well. That is a dashed white version of the top of a Lego. That is all we want to limit things to, because now the blue guy does not really know anything about the yellow guy, and the yellow guy does not really know anything about the blue guy, and they have both become simple.

So it is very important that you do not associate simplicity with partitioning and stratification. They do not imply it. They are enabled by it. If you make simple components, you can horizontally separate them, and you can vertically stratify them. But you can also do that with complex things, and you are going to get no benefits.

And so I would encourage you to be particularly careful not to be fooled by code organization. There are tons of libraries that look, oh, look, there is different classes; there are separate classes. They call each other in sort

Figure 3: 00:34:31 Modularity and Simplicity - build slide

of these nice ways. Then you get out in the field and you are like, oh, my God! This thing presumes that that thing never returns the number 17. What is that?

[Time 0:35:37]

```
slide title: State is Never Simple

+ Complects value and time
+ It _is_ easy, in the at-hand and familiar senses
+ Interweaves everything that touches it, directly or indirectly
  + Not mitigated by modules, encapsulation
+ Note - this has nothing to do with asynchrony
```

OK. I am not going to get up here and tell you state is awesome. I like state. I am not a functional whatever guy, whatever. I am going to say instead: I did this, and it sucked. I did years and years: C++, He-Man, stateful programming. It is really not fun. It is not good. It is never simple.

Having state in your program is never simple, because it has a fundamental complecting that goes on in its artifacts. It complects value and time. You do not have the ability to get a value independent of time. And sometimes not an ability to get a value in any proper sense at all.

But again, it is a great example. This is easy. It is totally familiar. It is at hand. It is in all the programming languages. This is so easy. This complexity is so easy.

And you cannot get rid of it. Everything – I have modularity. That assignment statement is inside a method. Well, if every time you call that method with the same arguments, you can get a different result, guess what happened? That complexity just leaked right out of there. It does not matter that you cannot see the variable.

If the thing that is wrapping it is stateful, and the thing that is wrapping that is still stateful, in other words by stateful I mean every time you ask it the same question you get a different answer, you have this complexity and it is like poison.

It is like dropping some dark liquid into a vase. It is just going to end up all over the place. The only time you can really get rid of it is when you put it inside something that is able to present a functional interface on the outside, a true functional interface: same input, same output. You cannot mitigate it through the ordinary code organization things.

And note in particular, I did not talk about concurrency here. This is not about concurrency. This has nothing to do with concurrency. It is about your ability to understand your program.

Your program was out there. It is single threaded. It did not work. All the tests passed. It made it through the type checker. Figure out what happened. If it is full of variables, what are you going to need to try to do? Recreate the state that was happening at the client when it went bad. Is that going to be easy? No!

[Time 0:38:00]

```
slide title: Not all refs/vars are Equal

+ None make state simple
+ All warn of state, help reduce it
+ Clojure and Haskell refs _compose_ value and time
  + Allow you to extract a simple value
  + Provide abstractions of time
+ Does your var do that?
```

But we fixed this, right? Your language, your new, shiny language has something called var, or maybe it has refs or references. None of these constructs make state simple. That is the first, primary thing. I do not want to say that even of Clojure's constructs. They do not make state simple in the case I am talking about, in the nature of simple I am talking about.

But they are not the same. They all do warn you when you have state, and that is great. Most people who are using a language where mutability is not the default and you have to go out of your way to get it, finds that the programs you end up writing have dramatically, like orders of magnitude, less state than they would otherwise, because they never needed all the other state in the first place. So that is really great.

But I will call out Clojure and Haskell's references as being particularly superior in dealing with this, because they compose values and time. There are actually little constructs that do two things. They have some abstraction over time and the ability to extract a value. That is really important, because that is your path back to simplicity. If I have a way to get out of this thing and get a value out, I can continue with my program. If I have to pass that variable to somebody else or a reference to something that is going to find the variable every time through the varying thing, I am poisoning the rest of my system. So look at the var in your language and ask if it does the same thing.

[Time 0:39:27]

```
slide title: The Complexity Toolkit

      Construct             Complects
--------------------  ----------------------------

        State         Everything that touches it

       Objects          State, identity, value
```

```
       Methods        Function and state, namespaces

        Syntax             Meaning, order

     Inheritance              Types

   Switch/matching       Multiple who/what pairs

     var(iable)s            Value, time

Imperative loops, fold       what/how

        Actors               what/who

         ORM                   OMG

     Conditionals       Why, rest of program
```

All right, let us see why things are complex. State, we already talked about. It complects everything it touches. Objects complect state, identity, and value. They mix these three things up in a way that you cannot extricate the parts.

Methods complect function and state, ordinarily. In addition, in some languages, they complect namespaces. Derive from two things in Java that have the same name method, and [hand gesture with sounds like explosion]. It does not work.

Syntax, interestingly, complects meaning and order, often in a very unidirectional way. Professor Sussman made the great point about data versus syntax, and it is super true. I do not care how much you really love the syntax of your favorite language. It is inferior to data in every way.

Inheritance complects types. These two types are complected. That is what it means: inheritance, complecting. It is definitional.

Switching and matching, they complect multiple pairs of who is going to do something and what happens, and they do it all in one place in a closed way. That is very bad.

Vars and variables, again, complect value and time, often in an inextricable way. You cannot obtain a value. We saw a picture during a keynote yesterday of this amazing memory where you could de-reference an address and get an object out. I want to get one of those computers. Have you ever used one of those computers? I cannot get one. I called Apple, and they were like, pff, no.

[Time 0:41:14]

The only thing you can ever get out of a memory address is a word, a scalar, the thing that was all derided. Recovering a composite object from an address is not something computers do, none of the ones that we have. So variables have the same problem. You cannot recover a composite mutable thing with one de-reference.

Loops and fold: loops are pretty obviously complecting what you are doing and how to do it. Fold is a little bit more subtle because it seems like this nice, somebody else is taking care of it. But it does have this implication about the order of things, this left to right bit.

Actors complect what is going to be done and who is going to do it.

[Audience laughter]

Now Professor Sussman said all these talks have acronyms, and I could not actually modify my slides in time, so object relational mapping has “oh, my God” complecting going on. You cannot even begin to talk about

how bad it is. And if you are going to do duals, what is the dual of value? Is it co-value? What is a co-value? It is an inconsistent thing. Who wants that?

And conditionals, I think, are interesting. This is sort of more cutting edge area. We have a bunch of sort of rules about what our programs are supposed to do. It is strewn all throughout the program. Can we fix that? Because that is complected with the structure of the program and the organization of the program.

[Time 0:42:55]

slide title: The Simplicity Toolkit

Construct	Get it via ...
Values	final, persistent collections
Functions	a.k.a. stateless methods
Namespaces	language support
Data	Maps, arrays, sets, XML, JSON etc
Polymorphism a la carte	Protocols, type classes
Managed refs	Clojure/Haskell refs
Set functions	Libraries
Queues	Libraries
Declarative data manipulation	SQL/LINQ/Datalog
Rules	Libraries, Prolog
Consistency	Transactions, values

So if you take away two things from this talk, one would be the difference between the words "simple" and "easy". The other, I would hope, would be the fact that we can create precisely the same programs we are creating right now with these tools of complexity, with *dramatically* drastically simpler tools. I did C++ for a long time. I did Java. I did C#. I know how to make big systems in those languages, and I completely believe you do not need all that complexity. You can write as sophisticated a system with dramatically simpler tools, which means you are going to be focusing on the system, what it is supposed to do, instead of all the gook that falls out of the constructs you are using.

So I would love to say the first step in getting a simpler life is to just choose simpler stuff. So if you want values, usually you can get it. Most languages have something like values. Final or val lets you declare something as being immutable. You do want to find some persistent collections because the harder thing in a lot of languages is getting aggregates that are values. You have got to find a good library for that, or use a language where that is the default. [intentional small cough]

[Audience laughter]

Functions, most languages have them. Thank goodness. If you do not know what they are, they are like stateless methods.

[Audience laughter]

Namespaces is something you really need the language to do for you and, unfortunately, it is not done very well in a lot of places.

Data: Please! We are programmers. We supposedly write data processing programs. There are all these programs that do not have any data in them. They have all these constructs we put around it and globbed on top of data.

Data is actually really simple. There are not a tremendous number of variations in the essential nature of data. There are maps. There are sets. There are linear, sequential things. There are not a lot of other conceptual categories of data. We create hundreds of thousands of variations that have nothing to do with the essence of this stuff, and make it hard to write programs that manipulate the essence of the stuff. We should just manipulate the essence of the stuff. It is not hard. It is simpler.

[Time 0:45:10]

Also, the same thing for communications. Are we all not glad we don't use the Unix method of communicating on the Web? Any arbitrary command string can be the argument list for your program, and any arbitrary set of characters can come out the other end. Let us all write parsers.

[Audience laughter]

No, I mean it is a problem. It is a source of complexity. So we can get rid of that. Just use data.

The biggest thing, I think, the most desirable thing, the most esoteric, this is tough to get, but boy when you have it your life is completely, totally different thing, is polymorphism a la carte. Clojure protocols and Haskell type classes, and constructs like that, give you the ability to independently say, "I have data structures. I have definitions of sets of functions, and I can connect them together." And those are three independent operations. In other words, the genericity is not tied to anything in particular. It is available a la carte. I do not know of a lot of library solutions for languages that do not have it.

I already talked about manage references and how to get them. Maybe you can use closures from different Java languages.

Set functions, you can get from libraries. Queues, you can get from libraries. You do not need special communication language.

You can get declarative data manipulation by using SQL or learning SQL, finally. Or something like LINQ, or something like Datalog.

I think these last couple of things are harder. We do not have a lot of ways to do this well-integrated with our languages, I think, currently. LINQ is an effort to do that.

Rules, declarative rule systems, instead of embedding a bunch of conditionals in our raw language at every point of decision. It is nice to sort of gather that stuff and put it over some place else. You can get rule systems in libraries, or you can use languages like Prolog.

If you want consistency, you need to use transactions, and you need to use values.

There are reasons why you might have to get off of this list, but, boy, there is no reason why you should not start with it.

[Time 0:47:20]

`slide title: Environmental Complexity`

```
+ Resources, e.g. memory, CPU
+ _Inherent_ complexity in implementation space
  + All components contend for them
+ Segmentation
  + waste
+ Individual policies don't compose
  + just make things more complex
```

OK. There is a source of complexity that is really difficult to deal with, and not your fault. I call it environmental complexity. Our programs end up running on machines next to other programs, next to other parts of themselves, and they contend for stuff: memory, CPU cycles, and things like that.

Everybody is contending for it. This is an inherent complexity. Inherent is Latin for "not your fault". In the implementation space and, no, this is not part of the problem, but it is part of the implementation. You cannot go back to the customer and say, "the thing you wanted is not good because I have GC problems." But the GC problems and stuff like that, they come into play.

There are not a lot of great solutions. You can do segmentation. You can say this is your memory, this is your memory, this is your memory, this is your CPU and your CPU. But there is tremendous waste in that, because you pre-allocate. You do not use everything. You do not have sort of dynamic nature.

But the problem I think we are facing, and it is not one for which I have a solution at the moment, is that the policies around this stuff do not compose. If everybody says, "I will just size my thread pool to be the number of . . . ". Of course. How many times can you do that in one program? Not a lot and have it still work out.

So, unfortunately, a lot of things like that, splitting that stuff up and making it an individual decision, is not actually making things simpler. It is making things complex because that is a decision that needs to be made by someone who has better information. And I do not think we have a lot of good sources for organizing those decisions in single places in our systems.

[Time 0:49:03]

```
slide:

Programming, when stripped of all its
circumstantial irrelevancies, boils down
to no more and no less than very
effective thinking so as to avoid
unmastered complexity, to very vigorous
separation of your many different
concerns.

                        Edsger W. Dijkstra
```

This is a hugely long quote. Basically, it says programming is not about typing, like this [gestures of typing on a keyboard]. It is about thinking.

[Time 0:49:18]

```
slide title: Abstraction for Simplicity

+ Abstract
  + _drawn away_
+ vs Abstraction as complexity _hiding_
+ Who, What, When, Where, Why and How
+ I don't know, I don't want to know
```

So the next phase here – I have got to move a little bit quicker – is how do we design simple things of our own? So the first part of making things simple is just to choose constructs that have simple artifacts. But we have to write our own constructs sometimes, so how do we abstract for simplicity? And abstract, again, here is an actual definition, not made up one. It means "to draw something away". And, in particular, it means to draw away from the physical nature of something.

I do want to distinguish this from, that sometimes people use this term, really grossly, to just mean "hiding stuff". That is not what abstraction is, and that is not going to help you in this space.

I cannot totally explain how this is done. It is really the job of designing, but one approach you can take is just to do: who, what, when, where, why, and how? If you just go through those things, and look at everything you are deciding to do and say, "What is the 'who' aspect of this? What is the 'what' aspect of it?" This can help you take stuff apart.

The other thing is to maintain this approach that says, "I don't know; I don't want to know." I once said that so often during a C++ course I was teaching that one of the students made me a shirt. It was a Booch diagram, because we did not have whatever it is now, the unified one. And every line just said that. That is what you want to do. You really just don't want to know.

[Time 0:50:39]

```
slide title: What

+ Operations
+ Form abstractions from related sets of functions
  + _Small_ sets
+ Represent with polymorphism constructs
+ Specify inputs, outputs, semantics
  + Use only values and other abstractions
+ Don't complect with:
  + How
```

All right, so what is "what"? "What" is the operations. "What" is what we want to accomplish. We are going to form abstractions by taking functions and, more particularly, sets of functions, and giving them names. In particular – and you are going to use whatever your language lets you use. If you only have interfaces, you will use that. If you have protocols or type classes, you will use those. All those things are in the category of the things you use to make sets of functions that are going to be abstractions. And they are really sets of specifications of functions.

The point I would like to get across today is just that they should be really small, much smaller than what we typically see. Java interfaces are huge, and the reason why they are huge is because Java does not have union types, so it is inconvenient to say, "this function takes something that does this and that and that". You have to make a "this and that and that" interface, so we see these giant interfaces. And the thing with those giant interfaces is that it is a lot harder to break up those programs, so you are going to represent them with your polymorphism constructs.

They are specifications. They are not actually the implementations. They should only use values and other abstractions in their definitions. So you are going to define interfaces or whatever, type classes, that only take interfaces and type classes, or values, and return them.

And the biggest problem you have when you are doing this part of design is if you complect this with "how". You can complect it with "how" by jamming them together and saying, "here is just a concrete function" instead of having an interface, or "here is a concrete class" instead of having an interface. You can also complect it with "how" more subtly by having some implication of the semantics of the function dictate how it is done. Fold is an example of that.

Strictly separating “what” from “how” is the key to making “how” somebody else’s problem. If you have done this really well, you can pawn off the work of “how” on somebody else. You can say, “database engine, you figure out how to do this thing” or, “logic engine, you figure out how to search for this.” I do not need to know.

[Time 0:52:41]

```
slide title: Who

+ Entities implementing abstractions
+ Build from subcomponents direct-injection style
  + Pursue _many_ subcomponents
    + e.g. policy
+ Don't complect with:
  + component details
  + other entities
```

“Who” is about data or entities. These are the things that our abstractions are going to be connected to eventually, depending on how your technology works. You want to build components up from subcomponents in a sort of direct injection style. You do not want to hardwire what the subcomponents are. You want to, as much as possible, take them as arguments because that is going to give you more programmatic flexibility in how you build things.

You should have probably many more subcomponents than you have. So you want really much smaller interfaces than you have, and you want to have more subcomponents than you probably are typically having. Because usually you have none. And then maybe you have one when you decide, “oh, I need to farm out policy”. If you go in saying, “this is a job, and I have done who, what, when, where, why,” and I found five components, do not feel bad. That is great. You are winning massively by doing that. Split out policy and stuff like that.

And the thing that you have to be aware of when you are building the definition of a thing from subcomponents is any of those kind of yellow thinking about blue, blue thinking about yellow kind of hidden detail dependencies. So you want to avoid that.

[Time 0:53:45]

```
slide title: How

+ Implementing logic
+ Connect to abstractions and entities via polymorphism constructs
+ Prefer abstractions that don't dictate _how_
  + Declarative tools
+ Don't complect with
  + anything
```

How things happen, this is the actual implementation code, the work of doing the job. You strictly want to connect these things together using those polymorphism constructs. That is the most powerful thing. Yeah, you can use a switch statement. You could use pattern matching. But it is glomming all this stuff together.

If you use one of these systems, you have an open polymorphism policy, and that is really powerful, especially if it is runtime open. But even if it is not, it is better than nothing.

And again, beware of abstractions that dictate “how” in some subtle way because, when you do that, you are nailing the person down the line who has to do the implementation. You are tying their hands. So the more declarative things are, the better things work.

“How” is sort of the bottom. Do not mix this up with anything else. All these implementations should be

islands as much as possible.

[Time 0:54:40]

```
slide title: When, Where

+ Strenuously avoid complecting these with anything in the design
+ Can seep in via directly connected objects
  + Use queues
```

When and where, this is pretty simple. I think you just have to strenuously avoid complecting this with anything. I see it accidentally coming in, mostly when people design systems with directly connected objects. So if you know your program is architected such that this thing deals with the input, and then this thing has to do the next part of the job. Well, if thing A calls thing B, you just complected it. And now you have a when and where thing, because now A has to know where B is in order to call B, and when that happens is whenever A does it.

Stick a queue in there. Queues are the way to just get rid of this problem. If you are not using queues extensively, you should be. You should start right away, like right after this talk.

[Audience laughter]

[Time 0:55:33]

```
slide title: Why

+ The policy and rules of the application
+ Often strewn everywhere
  + in conditionals
  + complected with control flow etc
+ Explore rules and declarative logic systems
```

And then there is the "why" part. This is sort of the policy and rules. I think this is hard for us. We typically put this stuff all over our application. Now if you ever have to talk to a customer about what the application does, it is really difficult to sit with them in source code and look at it.

Now, if you have one of these pretend testing systems that lets you write English strings so the customer can look at that, that is just silly. You should have code that does the work that somebody can look at, which means to try to put this stuff some place outside. Try to find a declarative system or a rule system that lets you do this work.

[Time 0:56:09]

```
slide title: Information _is_ Simple

+ Don't ruin it
+ By hiding it behind a micro-language
  + i.e. a class with information-specific methods
  + thwarts generic data composition
  + ties logic to representation du jour
+ Represent data as data
```

Finally, in this area, information: it is simple. The only thing you can possibly do with information is ruin it.

[Audience laughter]

Don't do it! Don't do this stuff. We have got objects. Objects were made to encapsulate IO devices, so there is a screen, but I cannot touch the screen, so I have the object. There is a mouse. I cannot touch the mouse,

so there is an object. That is all they are good for. They were never supposed to be applied to information. And we apply them to information. That is just wrong. It is wrong.

But I can now say it is wrong for a reason. It is wrong because it is complex. In particular, it ruins your ability to build generic data manipulation things. If you leave data alone, you can build things once that manipulate data, and you can reuse them all over the place, and you know they are right once and you are done.

The other thing about it, which also applies to ORM, is that it will tie your logic to representational things, which again tying, complecting, intertwining. So represent data is data. Please start using maps and sets directly. Don't feel like I have to write a class now because I have a new piece of information. That is just silly.

[Time 0:57:19]

```
slide:

Simplicity is not an objective in
art, but one achieves simplicity
despite one's self by entering
into the real sense of things

                  Constantin Brancusi
```

[Time 0:57:22]

```
slide title: Simplifying

+ Identifying individual threads / roles / dimensions
+ Following through the user story / code
+ Disentangling

[ several photos and drawings of knots ]
```

So the final aspect, so we choose simple tools. We write simple stuff. And then sometimes we have to simplify other people's stuff. In particular, we may have to simplify the problem space or some code that somebody else wrote. This is a whole separate talk I am not going to get into right now. But the job is essentially one of disentangling. We know what is complex. It is entangled. So what do we need to do? We need to somehow disentangle it.

You are going to get this. You are going to need to first sort of figure out where it is going. You are going to have to follow stuff around and eventually label everything. This is the start. This is roughly what the process is like. But again, it is a whole separate talk to try to talk about simplification.

All right, I am going to wrap up in a couple of slides.

[Time 0:58:13]

```
slide title: Simplicity is a Choice

+ Requires vigilance, sensibilities and care
+ Your sensibilities equating simplicity with ease and familiarity are wrong
  + Develop sensibilities around entanglement
+ Your 'reliability' tools (testing, refactoring, type systems) don't care
  + and are quite peripheral to producing good software
```

The bottom line is: simplicity is a choice. It is your fault if you do not have a simple system. And I think we have a culture of complexity. To the extent we all continue to use these tools that have complex outputs, we are just in a rut. We are just self-reinforcing. And we have to get out of that rut.

But again, like I said, if you are already saying, "I know this. I believe you. I already use something better. I have already used that whole right column," then hopefully this talk will give you the basis for talking with somebody else who does not believe you. Talk about simplicity versus complexity.

But it is a choice. It requires constant vigilance. We already saw that guardrails do not yield simplicity. They do not really help us here.

It requires sensibilities and care. Your sensibilities about simplicity being equal to ease of use are wrong. They are just simply wrong. We saw the definitions of simple and easy. They are completely different things. Easy is not simple.

You have to start developing sensibilities around entanglement. You have to have entanglement radar. You want to look at some software and say, ugh! Not that I do not like the names you used, or the shape of the code, or there was a semicolon. That is also important, too. But you want to start seeing complecting. You want to start seeing interconnections between things that could be independent. That is where you are going to get the most power.

All the reliability tools you have, since they are not about simplicity, they are all secondary. They just do not touch the core of this problem. They are safety nets, but they are nothing more than that.

[Time 0:59:52]

```
slide title: Simplicity _Made Easy_

+ Choose simple constructs over complexity-generating constructs
  + It's the artifacts, not the authoring
+ Create abstractions with simplicity as a basis
+ Simplify the problem space before you start
+ Simplicity often means making more things, not fewer
+ Reap the benefits!
```

So, how do we make simplicity easy? We are going to choose constructs with simpler artifacts and avoid constructs that have complex artifacts. It is the artifacts. It is not the authoring. As soon as you get in an argument with somebody about, oh, we should be using whatever, get that sorted out because, however they feel about the shape of the code they type in is independent from this. And this is the thing you have to live with.

We are going to try to create abstractions that have simplicity as a basis. We are going to spend a little time up front simplifying things before we get started.

And recognize that when you simplify things, you often end up with more things. Simplicity is not about counting. I would rather have more things hanging nice, straight down, not twisted together, than just a couple of things tied in a knot. And the beautiful thing about making them separate is you will have a lot more ability to change it, which is where I think the benefits lie.

So I think this is a big deal, and I hope everybody is able to bring it into practice or use this as a tool for convincing somebody else to do that.

[Time 1:01:12]

```
slide:

Simplicity is the ultimate sophistication.

              Leonardo da Vinci
```

So I will leave you with this. This is what you say when somebody tries to sell you a sophisticated type system.

[Audience laughter]

Thank you.

[Audience applause]

[Time 1:01:22]