Agility & Robustness: Clojure spec

- **Speaker: Stuart Halloway**
- **Conference: Strange Loop 2016 - September 2016**
- **Video: https://www.youtube.com/watch?v=VNTQ-M_uSo8**

[Time 0:00:00]

```
slide title: Agility & Robustness: Clojure spec

brought to you by
cognitect
```

Good morning, Strange Loop. My name is Stuart Halloway. Thanks for having me. I am really excited to be here.

And I am going to be talking about Clojure spec, and specifically about agility and robustness in programming, and how Clojure spec can help with that.

[Time 0:00:22]

```
slide title: My Daily Concerns

+ Cognitect founder
+ Clojure committer
+ Datomic developer
+ support team
+ cloud operations
```

And I want to start with a very brief background of the things that I worry about every day that shape, to some degree, the characteristics of spec, and why I want to use it.

I am the founder of a consulting company, Cognitect, originally Relevance. That is since 2003. So I have 13 years of building systems for other people, and moving from project to project.

I have been a Clojure committer since Clojure 1.0, and so I have had responsibility for a substantial open source project.

I am also a developer on the Datomic database. So commercial project development. So a lot of different kinds of software development.

I also run the support team at Cognitect. So I get to really eat my dog food. When we make software, whether it is consulting or open source or product software, and it is not working out great for people, I am where the buck stops, and we get to hear about it, which shapes your life a lot. You know, compared to my early days as a consultant where you just write your software and then you get to walk away, and somebody else has to worry about it.

And I am also responsible for operations for a lot of cloud software.

And so these things combine to have inculcated in me a real life cycle view of software that includes everything from before ideation to maintenance, and working on a really brown field.

[Time 0:01:56]

```
slide title: What I Want

+ correct: "free from error"
+ agile: "able to move quickly and easily"

+ robust: "able to withstand or overcome adverse conditions"
```

So when I think about these things, I want a lot of different things. And I am going to talk about three of them today.

I want programs that are correct. I want systems that are correct, that are free from error.

I want to be agile. I want to be able to move quickly and easily, modifying systems, changing them to meet needs.

And I want my systems to be robust. I want them to be able to withstand adverse conditions.

[Time 0:02:23]

```
slide title: How
```

Goal	Industry Practice
correct	types
	example tests
	encapsulation
agile	IDE refactoring
	local concision
robust	TBD?

And as an industry, we have a lot of techniques and language level practices, and other kinds of things that help us do this. And I cannot cover them all today.

But just a sampling of some that have been important to me. You see a lot of people using type systems, or example based testing, which a lot of people start with calling unit testing, to try to ensure that their programs are correct.

To be agile requires a lot of different things. Some people leverage encapsulation. So I know that I can make a change over here, and it is not going to cause the lights to turn off on that side of the building, because those systems are encapsulated from each other.

People also rely on IDEs that can automate refactoring, so you can take a large and complicated code base and move it without having to touch tens of thousands of lines of code yourself.

And people also have come increasingly, and I think correctly, to value what I am going to call local concision, which is that at the micro level, when you are working on a specific problem, you can answer that problem in three lines of code, or one line of code, instead of 30 lines of code spread across 10 modules. And if anybody, like me, worked with the first versions of Java's Enterprise JavaBeans, you will know exactly what I am talking about when you want to have local concision in order to be agile.

Robustness is a little bit more of a green field for us as software developers, so I am not going to sort of lay down established practices here. But you see people doing things like chaos engineering and fuzz testing and simulation testing and generative testing. But I think we have a lot of room to think about those things.

[Time 0:04:04]

```
slide title: Clojure Power
```

```
Goal     Industry Practice           Clojure Power
-------  -----------------        -------------------


                                    pure functions
correct       types                  state, flow
          example tests           systemic generality

         --encapsulation-- [strike-through]
agile     IDE refactoring     +       simplicity
          local concision         systemic generality

robust                              immutable data
                                  systemic generality
```

Clojure, as a language, does not change all of that. Clojure programmers do not say: don't want tests, don't want types, don't want those things. But Clojure adds a particular spin that helps out in all of these areas.

In terms of correctness, Clojure encourages programming with pure functions. And Clojure also has a distinctive model for state, called the unified update model in some circles, and a model for flow with channels.

And so these are all things that help you write correctly out the door. They help you write small things that you can compose about, and reason about.

For agility, Clojure's real buzzword is simplicity. If you make simple things, you will be able to take the Legos and recombine them a different way, whereas if everything is all fused together you get stuck.

And then at the language level there is not necessarily a huge amount of help for robustness, but immutable data makes a big difference. If you know that the data in your system never changes, it is much easier to walk up to a system that is in trouble and figure out what is going on. And I actually think this is probably the stealth advantage of functional programming, wearing my ops hat, is being able to walk up to a system and say: we are in big trouble here, but I know that 95% of the information in this system is just sitting there in RAM and never changes, because it is immutable. So I can immediately rule out the class of problems that come from: well when I looked at it a minute ago, it was like this, and if I come back and look at it again, it is going to be different.

And then I have this other term here, which I have put in all three rows, and am about to define, or characterize at least, which is: systemic generality.

[Time 0:05:59]

```
slide title: Systemic Generality

+ generality
  + all domains use the same general-purpose data structures and
    functions
+ systemic
  + in libs, in apps, in config, on wires, at rest
```

And so the idea with systemic generality is that all domains use the same general purpose data structures. Where in a lot of languages you would make up a new type, or make up a new class or a new object to represent every entity in the system, Clojure encourages you to represent entities as, for example, maps, or dictionaries.

And so you might walk up to a system written in C# or Ruby and find hundreds of classes representing different things in you domain, and in Clojure you are going to find one class. And you do not even see it as a class. You are going to find maps and you are going to program with them.

And the advantage of this is really summed up by the Perlis quote, which is: it is better to have a large number of functions that all work on a single data type, than to have a lot of disparate functions that work on specific data types.

And in Clojure, this generality is extended by being systemic. So when you write Clojure programs, you have the ability to use these generic data structures everywhere. You can use them in your code, but you can also use them in your configuration files, which could be written in edn.

[Time 0:07:10]

You can use them on the wire with edn or with Fressian. You can use them in your durable store.

And so what this does is: it gives you the ability to write really small code, and work in a very general way, and have the algorithms that do that job be much better vetted, because there is a small number of them. That is kind of a downstream consequence that people do not necessarily immediately think about. So instead of having 10,000 different functions in your system, you have 100. If you have 100, you can spend a lot more time making sure those 100 functions are correct. You can spend a lot more time making sure those 100 functions are fast.

And then, doing this across the system means that you do not have to have toll booths everywhere in your system where you are crossing a boundary from: is this my code, is this my configuration? How many people have written something that they are totally happy with from a unit testing perspective, or from a static typing perspective, and then it falls over in production because there is something busted in an XML file somewhere? Or something shows up in a bit of JSON that you did not expect?

So taking this systemic approach is really valuable. And Clojure has been around for over a decade now. And so we have some experience reports with what it is like to program like this.

[Time 0:08:26]

```
slide title: Pretty Good!

"A lot of the best programmers and the most productive programmers I
know are writing everything in Clojure and swearing by it, and then
just _producing ridiculously sophisticated things in a very short
time_.  And that programmer productivity matters."
```

https://thenewstack.io/the-new-stack-makers-adrian-cockcroft-on-sun-netflix-clojure-go-docker-and-more/

And the fact of it is: it is pretty good. People say great things. This is a quote that has been used in Clojure talks before talking about developers. The very best programmers are producing amazing things, and they are doing so really quickly.

So this is great. And if it were 2015, I would stop right now, and we could all hang out for a little bit and just chat and get to know each other. Or we could dig in and do a deep dive into the stuff that I just talked about. Talk about functional programming, or immutable data, or systemic generality.

But I want to do something totally different today, which is: talk about a problem that eats at the conscience of Clojure developers.

[Time 0:09:09]

```
slide title: Challenge

+ discipline required to deal with specificity

+ is systemic generality
  + scary?
  + a winning tradeoff?
  + winning?
```

And that problem is that this systemic generality idea is fantastic, except that we all work with specificity. When you sit down to actually work in your domain, you are actually thinking about addresses, or order items, or models of the human body, or models of equipment on a factory floor. They are actually specific things.

And so the Clojure promise is: your life is going to be better if you talk about those specific things in a really general way. And so what happens is: people have to be disciplined in how they think about specificity. Clojure does not have specificity as a language objective. It is kind of the opposite.

And so thinking about specific things is on you. And people go through two phases with that. If you are an outsider to Clojure, especially if you come from a world of static typing, it is just downright scary. Everything that you look at in your traditional programs that you are used to writing, that shows you how this domain is different from that domain, and what is interesting about your domains, appears to be gone in Clojure, in the name of this generality.

Then you use Clojure for a while and you come to conclude that this is very powerful, and it is delivering assistance – not perfection, but assistance – on writing correct programs, and being agile, and writing robust programs.

And then you start to try to justify Clojure to your friends as a winning tradeoff. We had to lose our ability to talk about specificity, and we get this tradeoff, and it is a clear win. It is 5 ounces of win, and one ounce of pain, or you will come up with your own numeric scoring system. But it seems like it is a big win.

And what I want to do is take the next step and say: there is a way to approach this where this does not have to feel like a tradeoff. Where it is all winning. And so you can program in the most generic way possible, and still deal with the specifics.

[Time 0:11:14]

```
slide title: Clojure spec

a standard, expressive, powerful, integrated system
for specification and testing
```

And that is what Clojure spec does. Clojure spec is a standard, powerful, integrated system in Clojure for specification and testing of your systems. And so this really answers the challenge of: how do I program generically and talk about specific things?

[Time 0:11:31]

```
slide title: spec Answers the Challenge

+ integrated language discipline for _a la carte_ specificity
  + without sacrificing generality
+ dynamic leverage
  + anytime
  + anywhere
  + up to you
```

And it answers it by saying not: you have to be specific everywhere, which is what static typing and object-oriented programming is sort of the direction they take you. But saying that you can be specific in an a la carte way, sort of running along side your actual code, sort of a separate place. And you can do this without sacrificing generality.

And then you can leverage the specificity at your discretion. So it is an entirely opt-in proposition. There are a bunch of different things that you can do with spec, or you can do no things with spec. Or you can not like spec, and keep programming Clojure without spec, and spec will not care. The bits are really insensitive. Their feelings will not be hurt if you choose not to use them.

[Time 0:12:17]

```
slide title: spec Power

Goal    Industry Practice     Clojure Power          spec Power
------- -----------------  -------------------  -------------------------


                             pure functions     specification, predicates
correct      types             state, flow        generative testing
         example tests     systemic generality      instrumentation

        --encapsulation-- [strike-through]         auto-documentation
agile    IDE refactoring +     simplicity     + explanation, conformance
         local concision   systemic generality       example data

robust                       immutable data          assertion
                           systemic generality      validation
```

And so when you think about Clojure and spec together, you get a ton of new power that helps with these objectives that I have. Because you have a way to do declarative specification, it is easier to look at a system and sort of convince yourself that you are moving in the direction of having it work correctly.

Also, you can automatically test your system with Clojure spec. So when you write your specs, you do not write tests, and spec writes your tests for you. If you have ever spent time maintaining a test suite, you should be very intrigued right now.

Also it allows you to instrument existing code at development time. So you can say: I want to run in a development mode where I enforce arbitrarily more complex programmatic validations of data and functions. And you can turn that on. And you can turn it on a little bit. You can turn it to 4, or you can turn it all the way up to 11. You can actually call “instrument-all”, which would instrument every single symbol in your system that had a spec.

On the agility side, a lot of being agile is being able to walk up to your system and understand it and manipulate it. And spec gives you several things here. It gives you documentation. So it adds to documentation automatically.

It gives you explanation. So when you have a piece of data, or a function, that you have used incorrectly, spec will tell you what is incorrect about that use. You could put that under the rubric: “good error messages”.

[Time 0:13:47]

On the other side of that, though, spec also provides conformance. And what conformance lets you do is say: do not tell me how my data was wrong, tell me how it was right. So conformance allows you to get information about the ways in which the data conforms to your domain. And oftentimes you are going to have, and we will see some examples, data that could be conformant to your needs in more than one way. So spec helps you understand what went wrong. It also helps you understand what went right.

And when you spec something, Clojure can generate example data for you. So there are a few people in the world – Rich Hickey for example – who think only in the abstract at all times, and never actually touch keyboards, and just manipulate things with their mind. But most of us benefit from examples. So one of the things that is fun about developing with spec is: as soon as you have made a spec, and it can even be a very rudimentary one, you can then say “give me example data” or “give me example functions that do this”.

[TBD - How can spec do the “give me example functions that do this” thing mentioned here?]

Finally, the tools that spec provides can be available any time. So they are not available only at compile time, like type checking. They are not available only at test time, like a test suite. And so you can have the same smarts that you use to understand how your system is put together, in effect at layer boundaries at run time, without writing new code.

How many people have had the experience of having individual modules work pretty well, but have to go and write a whole bunch of special stuff at the edges, just to make sure that things are happy? You get a lot of leverage for doing that with spec, because once you have written the spec, you can use the spec for that job.

So you are going to compare this to testing and static typing. And Twitter is going to have fun with it.

[Time 0:15:34]

```
slide title: Correct / Agile / Robust

[Color coding in this slide is indicated below in square brackets,
where green represents good, or at least better than yellow, and
yellow is better than red.]
```

	Example Tests	Types	Spec
expressive	[green] very	[yellow] varying	[green] very
powerful	[green] stakeholder correctness	[yellow] type correctness	[green] stakeholder correctness
integrated	[red] rare	[yellow] compile-time, must flow	[green] dynamic
specification	[red] no	[yellow] static	[green] yes
testing	[yellow] manual	[red] rare	[green] generative
agility	[red] expensive	[red] fragility	[green] dynamic
reach	[red] expensive	[red] libs, apps	[green] systemic

So here is your controversy slide, color coding and all. We will spend a few minutes on this, and then we will look at the actual use of spec, and look at some code.

So expressivity. Testing is very expressive, because you have the arbitrary power of your language at your disposal. It is as expressive as your language is. Type systems vary greatly in expressivity. A lot of industry code is written in things like Java and C#, which are not very expressive. And then people do cooler stuff with Haskell, or Liquid Haskell, or Idris, but on balance type systems have varying expressivity, and they certainly do not express things that you only know at run time, because they do not play at that time. spec is written in arbitrary Clojure code, so it has arbitrary expressivity just like testing does.

Example tests really allow you to aspire to stakeholder correctness. Is it correct in the sense that the person that paid me to write it is happy? And spec also does that. Type systems are gradually moving in this direction, but if type systems actually solved this problem, then as soon as it compiled, everybody's job would be over, and statically typed systems would not have bug databases. Because we would know that static type correctness was equivalent to "everybody was happy", and we could just move on. Clearly not true.

[Time 0:17:00]

Testing tends not to be integrated in the language. There are a couple of more recent languages that really do integrate testing approach at the language level, but for the most part we use testing tools that are sort of add-on. Types are definitely integrated in your language. In fact, they are kind of too integrated, and so cool work is happening now to sort of have types be less integrated with your language, so you can sort of opt-in and opt-out of them. But generally the actual industry experience with types is: you have to flow them around everywhere. So you do something over here, which causes a type problem over here, which cascades over here, and now you are over here, and that is called a yak shave. So specs are integrated, but they are integrated dynamically. So you can adopt them to the degree that you want, and where you want.

Example tests are not specifications. They tend to be sort of ad hoc things. Types very much are specifications. Type systems are specifications, although they are static in nature. And spec is obviously a specification, because we named it that, so it has to be, right? But they are specifications, and because they are dynamic, because they exist in the run time of your code, you can do sort of what you want to with them.

So testing with writing tests by hand, is writing tests by hand. Hence my use of the word "manual" in the table. If you are writing tests by hand, that is what you are doing. And then you have to babysit those tests, and you have to keep them working, or comment them out, or walk up to an existing system where the test suite has 300 tests that are still passing, and 1000 that are not, and try to figure out what to do about that.

[Time 0:18:30]

Type systems actually can be used to generate tests, but that is pretty rare. So that is a column where type systems, as used in industry, could really take off. You could actually use types to say: given these types, I want to automatically generate examples from these types, and test from them. So that happens some. It could happen more.

And in spec, it is the default. You write specs, and you can get generative tests for you, based on those specs.

So testing is promoted for agility, but when you take a systemic view of testing for agility, it is very expensive, because you know that you end up having to write a cascading set of tests. You have to write small tests. You have to write medium sized tests. You have to write tests that cover the boundaries. You find redundancies in the work that you are doing. So tests do help with agility. I am certainly not going to argue that we should stop writing example based tests tomorrow, but they are expensive to maintain.

[Time 0:19:30]

Type systems, as they are used, are fragile because of over-specification, and this goes back to the point about the general versus the specific. If you say really specific things all over your code base, then when you change something, that causes a pachinko effect where things just sort of bounce around through the code base, and you have to sort of update that.

With spec, you can spec as much as you want, or as little as you want. And you can enforce a la carte. So enforcement is entirely separate from making the statements to begin with.

And then reach. Again, example tests can reach everywhere. You could write unit tests that test your XML configuration file, but it is expensive. People tend to focus testing on certain parts of their system, and less energy in other places.

Type systems tend to reach only the code parts of your systems. You do not have type systems on wires, or in storage, or in configuration files, and things like that. And of course, because spec is built on top of Clojure, it has that kind of reach.

So if you are coming for the “why do I care?”, we are now done. This slide is really why we built spec and what we were trying to do with it. And then I am going to use the next 20 minutes to show you some code, and show you what it looks like using it.

[Time 0:20:46]

`slide title: spec Leverage`

What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	autodoc
examples please	sample generator
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion

So having written specs, what kinds of things can you get help with? You can find out: what are the building blocks of my system? Because you have this declarative specification.

You can ask what invariants hold about data or about functions. You can say: make checks for that, and turn those on. Validation.

You can get detailed explanations of what went wrong. You can get detailed explanations of what went right. You get automatic documentation. You get automatic example generation.

You get the ability to turn on instrumentation on spec’d functions at development time. This is a development time feature. I need to emphasize that. Turn on instrumentation that allows you to do small, medium, or large, and complex validations of your function calls, and making sure that they are being called correctly.

On the other side, you can make sure that your functions are implemented correctly by using automatic generative testing.

And then you can leave assertions in your production system that leverage this same thing. So unlike with types and tests, which tend to be kind of gone once you are in production, these things are still there, and you can opt-in to the exact amount of protection you want.

[Time 0:21:53]

`slide title: Clojure`

`[Clojure logo]`

Now in order to see all of this, you have to know Clojure. And in a long standing Strange Loop tradition, I am going to assume you know Clojure and skip past these slides. I am going to give this talk in other environments later, but for now I am just going to assume you know Clojure, because we have got a lot to cover.

[He quickly skips past about 4 or 5 slides that are not included here.]

[Time 0:22:15]

```
slide title: Declaration

[Several boxes colored green in the slide are marked [green] below.]

             What                              How
---------------------------------    ---------------------
what are the building blocks?        declarative structure  [green]
what invariants hold?                arbitrary predicates   [green]
how do I check?                           validation
what went wrong?                         explanation
what went right?                         conformance
docs please                                autodoc
examples please                        sample generator
am I using this right?                  instrumentation
is my code correct?                   generative testing
can I recombine pieces like this?          assertion
```

And we are going to talk about the actual building blocks here. So we are going to start with: what does it mean to make a specification?

[Time 0:22:23]

```
slide title: Structural Predicates

                classes
                   |
                   V
boolean?      bigdec?     any?
char?         keyword?    nil?
double?       ratio?      rational?  <-- crosscutting
int?          string?     some?
float?        symbol?     ident?
   ^          uuid?
   |
primitives
```

If you are a Clojure programmer, you already know how to make specifications, because all of the predicates in Clojure are specs. So these predicates cover things that you might think of as primitive-like, if you are looking at it from a Java language perspective. So things like: am I a long, or am I a double?

They also cover things that are class-like. So things like: am I a ratio, or a string, or a UUID?

But it also includes predicates that crosscut things that the Java type system underneath Clojure does not know anything about. So you can say "am I a rational number?", which is actually the union of types that do not have a type relationship between them.

[Time 0:23:00]

```
slide title: Arbitrary Predicates

true?         pos?   <---- value ranges
false?        int?
zero?         pos-int?      qualified-symbol?
```

```
#{0 1 2}      neg-int?      simple-symbol?
    ^         nat-int?            ^
    |                             |
    |                             |
explicit values           arbitrary runtime facts
```

But it goes further than this. Any arbitrary predicate can act as a spec. So predicates are not limited to talking about the shape of things. They can also talk about the run time values that they have. So you can have a spec that says: is something a specific value, like true, or false, or in the set 0, 1, and 2?

You can talk about the ranges of things. So is this a positive integer? Is this a negative integer?

And really you can talk about arbitrary run time facts. If you can write a function that takes an argument, and returns true or false, you are now participating in spec. You can write your own predicate, and that predicate is now a spec, and you can use it in all of these ways in your system.

[Time 0:23:37]

```
slide title: Collections

(s/coll-of string?)   <------- homogeneous

(s/coll-of int?
           :kind vector?
           :min-count 5      <----- size and type
           :max-count 10
           :distinct true)

(s/tuple int? int? keyword?)   <--- heterogeneous tuple

(s/map-of keyword?
          (s/coll-of int?))   <--- composition
```

Of course we have collections. And there is a rich set of support for collections, but just to give you a little bit of a flavor of it, you can say just: what is in my collection? So "coll-of string?". This is a collection that needs to be full of strings, and strings only.

And then there are a lot of bells and whistles. So you can say: I have a collection of integers, but that collection has to be of a vector shape. And there have to be at least 5 integers, :min-count 5, but there cannot be more than 10, so :max-count of 10. And by the way, all of the integers are :distinct. So the integer 2 cannot appear in there again.

You can have heterogeneous tuples. So this is a pain point sometimes in languages like Java where you really want these kind of generic things. You can just walk up and say: OK, this thing is an int, followed by an int, followed by a keyword.

And, of course everything composes. So you can say: I want to make a map of keywords to collections of integers. So this might be all of the different characters in a game, and then their scores. Or who knows?

[Time 0:24:40]

```
slide title: Boolean Logic

(s/and string?
       #(str/starts-with "SKU-" %))
```

```
(s/or :id pos-int?
      :email string?)
```

spec also has Boolean operators: “and” and “or”. So you can say: more than one thing has to be true about a piece of data. You can say: this thing has to be a string. But this is a string that represents a SKU in my catalog system, so it is a string, but it also has to start with “SKU-”.

Or you can have an “or”. So here we have a system where imagine that people can identify themselves either by their assigned numeric id, or by their email. You can say that a person’s identifier is either a positive integer or a string.

[Time 0:25:16]

```
slide title: Named Specs

(s/def :my.app/sku   <--- strong global names
  (s/and string?
         #(str/starts-with "SKU-" %)))

(s/def ::purchaser
  (s/or :account-id pos-int?
        :email string?))

(s/def ::import-line-item
  (s/tuple [::purchaser ::sku pos-int?]))
                 ^        ^
                 |        |
                 |        |
                by reference
```

Now once you start making these compositions in more elaborate specs, you very quickly want to have named specs. You want them to be a first class thing in the language so you can give them names. So the “def” form allows you to give specs names. You can say things like :my.app/sku is a string that starts with “SKU-”, and ::purchaser is someone who has either an account id, which is an int, or an email, which is a string.

And now I am going to be processing a CSV file. So when you use specs to describe an existing CSV file out in the world, which has 3 columns in it: it has a purchaser, which could be that integer or string. It has the SKU, which has to be a string that starts with “SKU-”. And a positive integer, which is the number of things of that SKU that that purchaser purchased, or whatever.

And the names here are global names. So these are namespaced names. The double colon in Clojure – here is your one little Clojure tidbit for the day – the double colon in Clojure is just a shorthand for “in the current namespace”. So this code is all running in the my.app namespace. So :my.app/sku could also be ::sku.

So the important thing there is that when you have named specs, then we can share specs across projects. You can build a spec that is composed of some specs that I wrote and some specs that are in Clojure. And all of those things can work together, because we are not going to have any name collisions.

[Time 0:26:36]

```
slide title: Syntax

_syntaxis_

"an arranging in order"
```

specs can also talk about syntax. So “syntax” is from the Greek. It is about the ordering of things. And if you are not actually a language implementer, you do not spend a lot of time necessarily thinking about the amount of time we as humans are using order to get meaning out of something.

[Time 0:26:57]

```
slide title: spec Syntax with Regexes

What                    How
-------------------     -----
order                   s/cat
choice                  s/alt
optionality             s/?
repetition              s/+
optional repetition     s/*
```

So spec has an integrated regular expression library.

It has catenation. So this comes in order, followed by that, followed by that. It has alternatives. So the “alt” form says: at this point in the syntax, you could have a this, or you could have a that. It has all of the things you expect to have in regular expressions.

But these are not regular expressions of strings. These are regular expressions of arbitrary data. So this is more like the computer science regular expressions than the Perl regular expressions that you might think of first when you hear that word.

[Time 0:27:27]

```
slide title: defn Syntax

(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

And so when you think about what that looks like, you can now start to spec all of the forms in the language itself. So this is a very small Clojure program that defines a function called “greet” that takes an argument called “your-name”, and then returns a string, which is “Hello,” followed by your name.

[Time 0:27:44]

```
slide title: specing defn

(defn greet     <-------------------> (s/cat :name simple-symbol?
  "Returns a friendly greeting"   <------->  :docstring (s/? string?)
                                             :meta (s/? map?)
  [your-name]   <------------------------->  :bs ::body-or-bodies)
  (str "Hello, " your-name))
```

The spec for that is a catenation of the name, which has to be a simple symbol, followed by a doc string, which is optional, so notice the question mark there. The doc string is optional.

Followed by some metadata. Well I do not even have any metadata in this example, so that was also optional.

And then the body part is actually quite complicated, and so you are taking advantage here of the fact that you can have named specs. So the :bs there name ::body-or-bodies. That thing is some complicated spec that describes what the shape of the internals of a Clojure function are.

[Time 0:28:16]

```
slide title: Maps as Information

(s/keys :req [::addr/name
              ::addr/street-1
strong ---->  ::addr/city      <---- critical decoupling:
global        ::addr/state           key _presence_ only
names         ::addr/zip]
        :opt [::addr/street-2])
```

You can spec maps as information. And this slide is essentially the replacement for OO, when you think of OO as sort of encapsulation, and entities, and getters and setters, and things like this.

So this is saying: I am going to talk about a map. But I am not going to make you make a new thing. You are going to keep passing maps around your program. And when you want to be specific, you are not going to stop using maps, you are just going to talk about your maps somewhere else with a spec like this.

And this spec says – and you can imagine that this is a spec for an address book entry – that an address book entry consists of a name and a street and a city and a state and a zip, and optionally a street-2.

Critically, here, this spec does not say what the shape of street or city or state are. Those things are specified elsewhere, which means this whole system composes. You can take this, and I do not even have to know what those things are.

In fact, those specs do not have to exist. You can absolutely use named specs that do not exist, up until the point where you have to program against them, and do something that wants them to exist. But just talking about things, you can talk about things and they do not even have to exist yet. So you can sort of build this in any order.

So that is all about spec'ing data.

[Time 0:29:28]

```
slide title: Functions

(str/index-of "pirate" "rat")
(str/index-of "pirate" \r     10)



(s/def ::args-for-index-of
  (s/cat :source string?
         :search (s/alt :string string?     <--- spec fn args
                        :char char?)
         :from (s/? nat-int?)))

(s/fdef my-index-of
        :args ::args-for-index-of
        :ret (s/nilable nat-int?))   <--- spec fn returns
```

But you can also spec functions. So when you spec a function, well what is a function? A function takes a list of arguments. Well that is ordering, so that is syntax. And a function returns something. That is a return value.

So here I have a function called "index-of" that takes a string, and something you are searching for, and maybe a start position, and you can spec that as a function. You can say: the args for index-of are a catenation –

any argument list for a function is a regular expression; it is a catenation of its arguments – it is a string, it is an alternative of either a string or a character, and then optionally a start position.

And then the return is a nil-able natural int. So my index-of returns the position of something that it found, or if it did not find a thing it returns nil. So this is pretty cool. This allows you not to just talk about data, but to talk about functions.

[Time 0:30:16]

```
slide title: Function Semantics

(s/fdef my-index-of
        :args ::args-for-index-of
        :ret (s/nilable nat-int?)
        :fn (s/or
             :not-found #(nil? (:ret %))    <--- not found :nil
  two -----> :found #(<= (:ret %)
categoric                (-> % :args :source count))))
outcomes                    ^
                            |
                     ret is bound by size of input source
```

But where it gets insanely cool is when you add the third element to functions, which are semantic specifications. So what are the semantics of a function? The semantics of a function are the relationship between its arguments and its return value. This is fundamentally a dynamic thing. You have to run the function to see.

And in this case, I am going to specify that index-of has two categoric outcomes. Either I can find something, or I can not find it. Because I have the "or" form, I can give those things names. And then I can say what each of those looks like.

If it is not found, the return had better be nil. And if it is found, the return had better be less than or equal to the size of the search string. That is a run time thing. So the search string happened to have 30 characters in it. If index-of returned 50, that is a buggy implementation. That could not possibly be true. So these are predicates of the actual behavior of the system.

Now, notice at this point that I have given you a painfully whirlwind tour of a bunch of different things that you can say with spec, but I have not actually said what happens in your program when you say all of these things. And the answer is: nothing. You say all of those things, and now you have a set of descriptions of data in your system, sitting on your shelf.

[Time 0:31:34]

`slide title: When?`

What	How	When?
what are the building blocks?	declarative structure	up to you
what invariants hold?	arbitrary predicates	up to you
how do I check?	validation	up to you
what went wrong?	explanation	up to you
what went right?	conformance	up to you
docs please	autodoc	autogen
examples please	sample generator	up to you
am I using this right?	instrumentation	up to you
is my code correct?	generative testing	up to you

```
can I recombine pieces like this?          assertion         up to you
```

And you can do stuff with those descriptions. When can you do stuff with those descriptions? It is absolutely up to you. Which of the things do you want to do with those descriptions? It is also absolutely up to you.

So we are going to dive in and take a look at them.

[Time 0:31:47]

```
slide title: Execution

[Several boxes colored green in the slide are marked [green] below.]

             What                               How
---------------------------------    ---------------------
what are the building blocks?        declarative structure
what invariants hold?                arbitrary predicates
how do I check?                           validation  [green]
what went wrong?                         explanation  [green]
what went right?                         conformance  [green]
docs please                                autodoc
examples please                        sample generator
am I using this right?                  instrumentation
is my code correct?                   generative testing
can I recombine pieces like this?          assertion
```

So we will start with validation.

[Time 0:31:49]

```
slide title: Validation

(s/valid?
 (s/coll-of keyword?)
 [:a :b :c "oops"])

=> false
```

So validation is through a call called “valid?” So “valid?” take a spec, and then some data, and it returns a Boolean telling you whether or not that data matches the spec.

So here I have a collection with :a, :b, :c, and “oops”, and one of those things is not like the others. “oops” is not a keyword, so “valid?” returns false.

Now like most Boolean functions, “valid?” cuts the universe in half, but it is not very exciting to look at on the slide. There is not a whole lot going on there. So what you want to know is not just that you are wrong, but what went wrong?

[Time 0:32:19]

```
slide title: Explanation

(s/explain (s/coll-of keyword?) [:a :b :c "oops"])
=> In: [3] val: "oops" fails predicate: keyword?

(s/explain-data (s/coll-of keyword?) [:a :b :c "oops"])
=> #:clojure.spec{:problems
```

```
                 ({:path [],
                   :pred keyword?,  <-- how was it wrong?
    what was -----> :val "oops",
      wrong?        :via [],
                   :in [3]})}   <-- at what position
                                     was it wrong?
```

So you can also say: explain to me the way in which this data does not match my spec. When you ask for an explanation, you get: what was wrong. The value “oops” was the problem. :a, :b, and :c were fine. You get how it was wrong: the predicate was “keyword?”, which you did not match. You get what position it was wrong at. It is at position 3. 0, 1, 2, 3. We are 0-based by some weird tradition that started in the 1970s.

So now you have a detailed and precise pointer to where your problem happened. And by the way, this detailed and precise pointer, this is all compositional. So if you have the JSON ball of mud from hell that has got something wrong with it 17 layers deep in that thing, then you are going to have a path through that thing, and a thing pointing you exactly where the problem happened.

So it is nice to know how things went wrong,

[Time 0:33:13]

```
slide title: Conformance

(s/conform
 :clojure.core.specs/defn-args
 '(greet
    "Returns a friendly greeting"
    [your-name]  <-------------------------+
    (str "Hello, " your-name)))            | _how_ the value
                                           |    matched
{:name greet,                              |
 :docstring "Returns a friendly greeting",  V
 :bs [:arity-1 {:args {:args [[:sym your-name]]},
                :body [:body [(str "Hello, " your-name)]]}]}
```

but you might also want to know how things went right. So you can say “conform”. “conform” takes a spec and some data. In this case this data is actually program code. And it tells you, and you can follow the red here, the red line, it tells you how the value matched.

[The red line refers to the one beginning with :bs]

Well the symbol “your-name” was in the :bs branch of the “or”. It was in the “:arity-1” branch of another “or”. It was in the :args branch of the :args branch, with a :sym branch. Every one of those 5 things in red represents a point at which this thing could have been a valid function, but looked different. And the spec for “defn” is a pretty rich thing. There are a lot of different things that happen that are legal in “defn”. So it is a complicated spec.

And conformance gives you the ability to say: exactly how does this data conform? This, by the way, also gets rid of a ton of ad hoc parsing in your own program. If you are having to interpret something that is ordered, the first thing you want to do is often give names to that order. Well spec gives names to that order, and then you can call it on an a la carte basis.

If you are an experienced Clojure programmer, think of this as destructuring on steroids. This is data driven destructuring where you can drive the destructuring anywhere you want.

[Time 0:34:23]

```
slide title: Dev Assistance

[Several boxes colored green in the slide are marked [green] below.]

             What                              How
---------------------------------    ---------------------
what are the building blocks?        declarative structure
what invariants hold?                arbitrary predicates
how do I check?                           validation
what went wrong?                         explanation
what went right?                         conformance
docs please                                autodoc       [green]
examples please                        sample generator  [green]
am I using this right?                  instrumentation
is my code correct?                   generative testing
can I recombine pieces like this?          assertion
```

Of course you get documentation and example data.

[Time 0:34:26]

```
slide title: Autodoc

(s/fdef letter-grade
        :args (s/cat :n ::grade)
        :ret #{:A :B :C :D :F})

(doc letter-grade)
-------------------------
user/letter-grade
([n])
Spec
  args: (cat :n :user/grade)
  ret: #{:A :F :D :B :C}
```

So having spec'd that letter-grade is a function that takes a numeric grade, and returns one of the letters A, B, C, D, F, then in the documentation string you get a spec addendum section that says: here is what this is.

I have found that a lot of times, once I start spec'ing functions, I do not put anything else in the doc string. I just let the doc string get the little spec addendum, and that is all I need to know.

[Time 0:34:48]

```
slide title: Example Data

(s/def ::grade (s/int-in 0 100))
(s/exercise ::grade 25)

=> ([0 0] [1 1] [1 1] [1 1] [3 3]
    [9 9] [0 0] [5 5] [3 3] [8 8]
    [1 1] [98 98] [6 6] [6 6] [4 4]
    [50 50] [26 26] [63 63] [1 1] [69 69]
    [61 61] [63 63] [60 60] [71 71] [7 7])
     ^   ^
     |   |
     |  conformed value
     |
   value
```

You can also get sample data. So you can say: well I am going to define that a grade is a number. This is American. At least the grading system that I had when I was in high school. A grade is a number between 0 and 100. And then I can say "exercise" grade for me.

When you ask Clojure to exercise a spec, it will generate examples that conform to that spec, and show you how they conform. So the reason that you see pairs here is that the first piece of data is the data that it generated, and the second piece is how it conformed. And when you are conforming as a number, how it conformed is not very interesting, because it can only do one thing, so there is nothing in that second value that is interesting in that case.

[Time 0:35:24]

```
slide title: Example Fn Invocations

(s/exercise-fn #'letter-grade 25)

=> ([(0) :F] [(0) :F] [(1) :F] [(2) :F] [(0) :F]
    [(0) :F] [(0) :F] [(10) :F] [(1) :F] [(3) :F]
    [(52) :F] [(1) :F] [(0) :F] [(2) :F] [(11) :F]
    [(26) :F] [(60) :D] [(60) :D] [(61) :D] [(68) :D]
    [(94) :A] [(52) :F] [(63) :D] [(7) :F] [(50) :F])
                ^    ^
                |    |
             args   return
```

You can also, if you have spec'd a function, exercise that function. So here there is some function letter-grade that takes a numeric grade and returns a letter. I can say exercise this function for me. And spec can generate data that represent inputs to the function, call the function for you as many times as you want, and then return the output that results.

And you could eyeball this and go: yes, 94 should be an :A, and 60 should be a :D. That all looks pretty good.

Well once you have this ability to generate data, including function calls, in your system

[Time 0:35:55]

```
slide title: Robustness

[Several boxes colored green in the slide are marked [green] below.]

             What                              How
---------------------------------    ---------------------
what are the building blocks?        declarative structure
what invariants hold?                arbitrary predicates
how do I check?                           validation
what went wrong?                         explanation
what went right?                         conformance
docs please                                autodoc
examples please                        sample generator
am I using this right?                  instrumentation   [green]
```

```
is my code correct?                    generative testing  [green]
can I recombine pieces like this?          assertion      [green]
```

you have access to some powerful capabilities for making your system more robust.

[Time 0:36:01]

```
slide title: Instrumentation

(test/instrument `start-server)      <---- dev-time switch
(start-server {:host "localhost" :port :default})

ExceptionInfo Call to #'user/start-server did not conform to spec:
In: [0 :port] val: :default fails spec: :user/port
     ^        at: [:args :endpoint :port] predicate: pos-int?
:clojure.spec/args  ({:host "localhost", :port :default})   ^
:clojure.spec/failure  :instrument                          |
:clojure.spec.test/caller  {:file "example.clj",            |
     |                 +--> :line 160,                      |
     |                 |    :var-scope user/eval1552}       |
 position of           |                                what spec
  bad value       pinpoint invalid call                  failed
```

You can instrument your system. So here is a scenario where I have a development time switch that says “start-server” is going to be instrumented. I do not even know how it works. And then I am going to call it. I am going to call it incorrectly. It turns out :port :default is not legal. When instrumentation is turned on, I am going to get a specific error that says :port was the problem, that this spec failed, that the problem happened in this line of this module of my program.

And this is not a static thing. This is a running thing. This is not: you are limited to what you can do at static time. This is you are running your program and you can turn this up to 11 and say: something is going wrong here. I do not understand how this other subsystem is using what I am calling. Maybe I am not calling it correctly.

[Time 0:36:46]

```
slide title: Generative Testing

(->> (test/check `index-of)     <---- generate a huge
     test/summarize-results)         number of test cases

{:spec ...,
 :sym user/index-of,
 :failure
 {:clojure.spec/problems
  [{:path [:ret], :pred nat-int?,   <--- find problem
    :val nil, :via [], :in []}],
  :clojure.spec.test/args ("" "0"),  <--- shrink test case
  :clojure.spec.test/val nil,             to smallest repro
  :clojure.spec/failure :check-failed}}
```

Similarly, you can test automatically. So you can say: please check this for me. When you ask spec to check something for you, it is going to say: generate an arbitrary number of inputs, until you get tired. I think the default is 100. It is going to invoke the function for you. And then it is going to check the return predicate to make sure the thing is structurally sound.

And then it is going to check the function predicate to make sure that the thing's semantics are sound. And if it finds a problem, it is going to shrink to a small repro before it goes back to you. So it is going to find some big hairy example that shows that your program is broken, and then it is going to heuristically shrink that down to a tiny example.

So here it found a bug in a deliberately buggy program, with a really tiny argument set of the empty string, and the string "0".

[Time 0:37:33]

```
slide title: Assertion

(s/check-asserts true)     <--- turn on spec assertion
(s/assert
 (s/coll-of (s/int-in 1 7))   <--- validate against spec
 [6 6 9])

ExceptionInfo Spec assertion failed
In: [2] val: 9 fails predicate: (int-in-range? 1 7 %)
:clojure.spec/failure  :assertion-failed
                ^
                |
          precise errors
```

And of course you have assertions, where you can take these same detailed error messages, and sprinkle them where you want to in your production system. So you do not put them everywhere, but maybe on certain boundaries in your system, or certain internal places where you have presumptions. You can turn on assertion. You can leave assertions in your code. If "check-asserts" is off, there is no run time footprint.

[Time 0:37:54]

```
slide title: Power

[Entire "When" column of table boxes are colored green in the slide.]
```

What	How	When
what are the building blocks?	declarative structure	up to you
what invariants hold?	arbitrary predicates	up to you
how do I check?	validation	up to you
what went wrong?	explanation	up to you
what went right?	conformance	up to you
docs please	autodoc	automatic
examples please	sample generator	up to you
am I using this right?	instrumentation	up to you
is my code correct?	generative testing	up to you
can I recombine pieces like this?	assertion	up to you

So in summary, spec is about doubling down on being a dynamic language. Specification is a problem, but you do not need to be more static to solve it. Specification is a dynamic idea. It is something that allows you to deliver simple specifications separate from your code, and then pour in value in validation, in error checking, conformance, in testing, and add exactly the parts of value that you want.

[Time 0:38:26]

```
slide title: Experience Report

+ early days still
+ I already want this --everywhere-- [strike-through] anywhere
  + pro re nata  [Latin for "in the circumstances" or
                  "as the circumstance arises"]
+ be thoughtful about
  + overspecification
  + generation
```

So where are we with this? We are on Clojure alpha12 of Clojure 1.9. So we have done 12 alphas here. I expect we will go beta soon. So this will be beta soon. People are already using it in production, but it is early days. We are eager to get people's experience reports. There is just a ton of really cool stuff that people at Strange Loop are doing, and we would like to know how it works with spec.

Having said that, take advantage of the dynamic nature of spec. This is not an all or nothing thing that you have to jump into with both feet and do everywhere. Try using little bits of it here and little bits of it there.

Be thoughtful about the danger areas. And one danger area is definitely overspecification. spec has tried very hard to be as dynamic as possible, and as flexible as possible, and to not require you to make overly-specific statements. But, you can make whatever statements you want. It is a general purpose programming language. So I predict five years from now I will have consulting work going in and helping people make their specs more general, and keep things as general as they can. So that is definitely a challenge area.

Also generating data. There are a lot of interesting things to explore there, which we do not have time to do today.

[Time 0:39:37]

```
slide:

[ Clojure logo ]

clojure.org/about/spec
```

If you want to learn more about Clojure spec, you can see the web site clojure.org/about/spec, and I will be out in the hall. We do not have time for questions here, but I will be out in the hall for the next couple of hours, and if you are interested in talking more, I would love to hear from you.

[Time 0:39:50]

```
slide:

[ Cognitect logo ]

cognitect.com
```

My name is Stuart Halloway, and that was Clojure spec. Thank you.

[Audience applause]

[Time 0:39:53]