REPL-Driven Development

- **Speaker: Stuart Halloway**
- **Date: June 2017**
- **Video: https://vimeo.com/223309989**

```
slide title: REPL-Driven Development


brought to you by
cognitect
Stuart Halloway   Founder & President @ Cognitect
```

So my name is Stuart Halloway. I work at Cognitect, which is the company behind Clojure and ClojureScript and Datomic. I want to talk to you tonight about REPL-driven development. And that is there to be a catchy buzzword. I actually don't really believe in REPL-driven development any more than any other kind of driven development.

But I do want to talk about the importance of the REPL to a Clojure programmer. And I think this is an area where we have insufficiently lived up to the Lisp tradition. We treat the REPL sort of casually as a shell where we can do things. And I think we can do a lot more, in general, with it.

[Time 0:00:50]

```
slide title: Plan


Why REPL
How REPL
REPL-Driven Development
```

So the basic structure of the evening is three pieces.

The first piece is "Why?" Why is the REPL important? Why is it a distinguishing characteristic of a Lisp?

Then how? So mechanically. So there is this thing, the REPL. Obviously you can launch a REPL. How many people here have launched a REPL before? [most or all audience members raise their hand] So a good bet there.

So you can launch a REPL, but there is more to it than that. There are REPLs, and there are REPLs. And we are going to have a conversation about the latter kind.

And then we will talk about what it is like to drive from the REPL as you are developing.

[Time 0:01:28]

```
slide title: What is Clojure?


"First and foremost, Clojure is dynamic"


    FEATURES
    Dynamic Development
    Functional Programming
    Lisp
    Runtime Polymorphism
    Concurrent Programming
    JVM Hosted
```

https://clojure.org/about/dynamic

So it is interesting. If you go to look at the Clojure web site, and you look, this is the feature menu.

I know that the fonts are small. I do not have a ton of code or really small font stuff in this slide deck, so hopefully those of you in the back will survive this experience optically.

But if you go to the web site, you will see that on the feature list for Clojure it lists "dynamic development" first. These are not in alphabetical order. It is listed ahead of being a functional language, ahead of being a Lisp, ahead of running on the JVM. All of these other characteristics that people associate with Clojure.

And dynamic in this sense does not mean dynamically typed. It includes that, probably, although I could imagine statically typed languages providing this kind of dynamic experience.

But what it really means is: it is about having this dynamic experience where you are working in an environment where the run-time is tangible. You can walk up to the run-time programmatically and change pieces of it and reflect against it. Reflect against it in a read or a write way. Make it different. And that gives you an incredible feedback loop. So that is what we are talking about.

[Time 0:02:40]

```
slide title: Team Java Agrees

_Immediate feedback_ is important when learning a programming
language.  The number one reason schools cite for moving away from
Java as a teaching language is that other languages have a "REPL"
and have far lower bars to an initial "Hello, world!" program.
```

http://openjdk.java.net/jeps/222

And of course everybody agrees. If you go and look at the Java proposal for a shell in Java 9 – it is coming in Java 9, right?

Audience member: [hard to understand]

What's that?

Audience member: Delayed, but eventually it will be released.

It will eventually get there.

One of the arguments that is made for why Java is adding this is that the number one reason that schools move away from Java as a teaching language is the sort of the high bar against the "Hello, world!" program. There is a lot of ceremony and setup. You want to sit down at the computer and start typing and get the computer to respond to you.

[Time 0:03:17]

```
slide title: Read, Eval, Print, Loop

read: input stream -> data

eval: data -> data

print: data -> output stream
```

So the REPL is a read, eval, print loop.

Read is a function of an input stream that returns data. So it is a stream of characters to data.

Eval is a function of data to data. Right? It takes some data, and it produces some other data. And, on exciting days, also side effects. Launches missiles, or moves money, or whatever.

And then print is a function from data to an output stream, to a stream of characters.

And so this cycle within the REPL, within this loop, also provides a way to sort of round-trip information. Right? Information starts as characters in a human's head, or anywhere, on a network wire, or on storage. It is processed and then returned back to that stream of character states.

[Time 0:04:06]

```
slide title: REPL Advantages

Immediate interaction

  "Faster than a speeding test"

Interact with running programs

No "pour concrete" phase

  Copy code from a REPL dev session to your program, or vice versa
```

And there are several advantages to this. One of them is: you have this immediate interaction. REPL development is faster than test driven development. Full stop. It is. It just is. If you like test driven development because of the rapid feedback loop, REPL driven development is a faster feedback loop, because you do not have to write the test part. You can try things out.

Now this does not mean you should stop doing TDD, or stop writing tests, because tests have other advantages. Tests can be run again later. So there are reasons to do it. But if you are *just* talking about the feedback loop, you can really not beat the REPL.

You can also use the REPL to interact with running programs. And this is one area where already I think that people sort of underutilize the REPL. They think of the REPL as a sidecar on their development process, as opposed to, perhaps, the primary aspect of their development process.

It is also something that you can launch anywhere. So you can take a running program, that you are trying to understand what it is doing, attach a REPL to it while it is running, and just start interacting with it.

Or, you can attach more than one. Attach two REPLs, or five REPLs, to a running program. And there are reasons that you might want to do that as well.

And this style of programming really reduces the friction between development and production. And in fact, it reduces it, kind of, to you get to choose to introduce friction, which you probably should. There probably should be a little bit of friction between development and production. But depending upon what you are making, maybe not. Maybe every line you type goes straight into production, after you type it in.

But that boundary is really permeable. It is quite different from the "pour concrete" that people come to expect from less agile ways of approaching things.

[Time 0:05:49]

```
slide title: Shell vs. REPL

                    Shell             REPL

semantics           caveats           like programs

state               new abstractions  like programs
```

```
context              new wrappers      like programs

modifying code       new semantics     like programs

forward reference    new semantics     no (like programs!)

dependencies         new semantics     like programs

testing              new semantics     (should be) like programs

risks                classloaders confusing   <- yeah, that
```

http://openjdk.java.net/jeps/222

So it really is: the REPL comes first, and then there are other things built on top of it. And you can really see this difference when you look at Java's new shell. Java has this new shell. It is coming. And when you look at Java's shell, what are the semantics of the shell?

Well the shell has a bunch of caveats. The REPL is how we think about Clojure, but Java's shell. Well what about state? What if I want to do something stateful at the REPL? Well I can just change things. What if I want to make a new thing at the Java shell? Where does it go? The original language does not even have a place for that. So trying to backfill and put a shell in, now it is like "we have to make up a place". So we have to make up what "state" means.

What "state" means in a Clojure REPL is exactly what it means in your program. It is unified with your production experience.

Modifying code. If I want to modify code at the shell, we have to have rules for what it means to modify code, because Java originated as wanting to be a compiled thing where you have units of projects, and chunks of code, and classes, and classloaders. In the REPL, your program is already like a REPL interaction. Your program would not be different if you took all of your namespaces and pasted them together, in the right order, and just poured them through the REPL. You get the same thing out as going through projects and files. Projects and files, and the namespace to file name convention and all of that stuff. All of that is a convention that is not necessary at all, and it is important to keep that in mind.

What about forward references? Well, Java has to make up rules for forward references in their shell, because Java has forward references. Clojure says: well, we do not have forward references. I mean you can "declare" things ahead of using them, but you just do that. And you do it the same way you would in a program. There is not a special sort of shell thing.

Dependencies. The shell has new rules about that.

Testing things. The shell has new rules about that. And you can read the Java proposal. And the Java proposal is not a bad proposal. It is not technically illiterate or incompetent. It is just dealing with the fact that Java does not have a set of rules about how these things work at the bottom, and so they had to make some up. And so the shell experience is an imperfect reflection of a running program experience.

And then finally I will mention that they call out in the description of the shell that because you are dynamically reloading code, classloaders could become confusing. That is in fact also true with the Clojure REPL because we are unified with the host platform in terms of how we think about those things. So Clojure has dynamic classloaders, but it does not have a replacement for classloaders. And so classloader confusion is something that cannot be solved at this level.

But I could imagine if you were coming from Common Lisp, or something like that, and saying "that seems like a wart." And it is a wart that we lovingly embrace, for the mighty power of having access to the whole JVM ecosystem.

[Time 0:08:48]

```
slide title: Piling On

Tangible runtime

  Strong, reified names: symbols, keywords, and vars

  Reflection as data

Extensible data: edn

Integrated specification: clojure.spec

Inline assembler: Java
```

So on top of all that, though, the REPL has a lot more that distinguishes it from just a shell.

Clojure has a tangible runtime. You can walk up to it and say "what are the pieces of my program made of?" And so a program is made of namespaces. Namespaces contain vars. And vars point to functions or data, and you can reflect against all of those things.

All of those things have strong names. Clojure has strong names everywhere. If you did not have strong names everywhere, and you had to retrofit it, you might have the JavaScript ecosystem, as people have tried to make up idioms to sort of introduce a convention for making strong names.

The reflection itself is exposed as data, so it is easily manipulated, unlike Java reflection. When you do Java reflection you do not get back just raw Java data. You get special things: java.lang.Method, java.lang.Field, and so forth.

Then we have extensible data. So when we want to talk about things at the REPL, we cannot anticipate everything somebody else is going to want to talk about. And so the extensible data notation gives you the way to say: I want to extend what we can talk about with data literals.

And we have now, coming in 1.9, integrated specification. So you have ways to describe your data interactively at the REPL, as you go.

And of course we have this wonderful in-line assembly language, called Java. So instead of thinking about, "Why is that Java showing in my Clojure program?" You should be bragging about "Look at this almost incredibly seamless in-line assembler we have, that lets us call legacy code that 9 million developers around the world maintain for us." And we are very thankful that they do.

[Time 0:10:35]

```
slide title: Superstructure

Paredit

Syntax coloring

IDE integration

Debugger
```

On top of that, there is other stuff that people care about a lot. But this is not the phenomenon. This is the epiphenomenon. These things build on top of the other things.

And just to sort of put a stick in the ground, I have actually sorted them in the order that I think they are important to a developer. I think once you are looking above the raw tools, the one thing you need as a Clojure developer is Paredit. Something that is aware of the structure of your program, not just the text of your program.

And then, syntax coloring would sure be awful nice.

And if you have both of those things, we associate those things with IDEs, anyway. It would be nice to have an IDE, and a debugger would be OK, too. And happily we are now at a place with Clojure where you have things like Cursive.

[Time 0:11:15]

`slide:`

`[Screen shot of Cursive development window.]`

https://cursive-ide.com/userguide/navigation.html

And so Cursive delivers a first class IDE experience, where first class is defined as the best that people have come to expect in the Java ecosystem. And that is great. So if someone is coming to Clojure from Java, or C#, or whatever, then they can pick up Cursive, and they can have an experience where the IDE side feels kind of like what they expect.

That is not what this talk is about.

[Time 0:11:49]

`slide:`

`How to Use the REPL`

So you can use the REPL from an IDE, but I really want to focus on using the REPL itself, and how this works at a lower level than IDEs.

[Time 0:11:56]

`slide title: Using the REPL`

`Hook into a REPL`

`Socket REPL`

`Start more than one REPL`

`  Nest a subREPL`

`  Launch REPL at a point of interest`

`Launch UI elements from a REPL`

And so I want to point out a couple of things.

The first one is that you can hook into a REPL. So you do not have to accept the semantics of read, or eval, or print. You can override any of them. And in fact there are more things than that you can override, as we will see in a moment. So you can hook into a REPL.

There is, as of Clojure 1.8, there is the socket REPL. So the socket REPL is just the raw REPL exposed over a socket, which is one way that you could connect to a running program.

You can start more than one REPL. And where this comes up is when you start building tooling. Probably the right way to build GUI tooling on top of a REPL is not to make a fancy meta-protocol on top of REPL interaction, like nREPL. Rather, you want to have a plain REPL, and then another plain REPL that acts as the control channel to the GUI tool. And those REPLs can talk to each other. So unfortunately, things got built out of order. So nREPL got going and has a bunch of traction, and whatever.

But in a happier world, IDE tooling would be built on top of, hey, you start a plain old REPL. Nothing special. No additional libraries required. And then if the tooling needs to do special things like it wants to know what you are doing, so it can do completion, or whatever, you start a second REPL that is a backchannel just for the purpose of supporting the tooling. So that is an idea of why you might want to start more than one REPL.

And then another thing that people do not do enough of is launching user interface elements from a REPL. If you think of the REPL as foundational, then you can say, "pop me up a visualization of the last thing I did." Or "pop me up a visualization that is going to watch this atom as it changes." Or something like that.

Instead of taking the GUI-first approach, where you start an IDE and then inside of the IDE you have the REPL as the sort of the sidecar, I would rather have the GUI elements be the sidecar.

[Time 0:13:49]

```
slide title: Hook Into REPL
```

```
 1 (defn oops
 2   [^Throwable x]
 3   (println "An error occurred:" (.getMessage x))
 4   (println "For complete details, (pp/pprint *e)"))
 5
 6 (main/repl :caught oops)
 7
 8 (/ 1 0)
 9 An error occurred:  Divide by zero
10 For complete details, (pp/pprint *e)
```

So hooking into a REPL looks like this. I am going to define a function called "oops". "oops" takes an exception, and it prints out the message with the exception, and then a helpful instruction for how to get additional information about the error, which is "if you want more details, then you can call pretty-print on the exception."

I am then going to start a new REPL. So clojure.main/repl will start a new REPL. So in the middle of my REPL, I am just going to start another one. And in that REPL, I am going to replace the "caught" handler, because this is one of the things you can override: what happens when an exception occurs?

And then I am going to do something that causes an exception. I am going to divide 1 by 0, which is a divide by 0 arithmetic exception. And I am going to get back the message that I added to the caught handler. Given that we have this technology at the REPL, what is the right answer for making the Clojure beginner experience better in terms of error messages?

Let me ask a slightly different question. Given that the REPL works this way, does Clojure need to incorporate a bunch of changes in its 1.9 release cycle in order to provide a better experience for beginners?

No. Who can make a better experience for beginners? Anybody! Anybody can. If you have a particular error message that just absolutely chaps your butt, then run under a REPL that does something different when that error happens.

And by the way, if you wanted to dispatch on errors and do different things, do we have powerful tools for polymorphic dispatch in Clojure? Hugely powerful. What do we have? We have protocols. We have multimethods. We have spec. You could use spec to look at the information content of an error, and say ... Oh! And you could pop up Clippy. You could pop up a little paper clip that says, "Oh, I see you are trying to use infix notation. Clojure is a prefix language." Or something like that.

[Time 0:15:52]

slide title: Socket REPL

```
1 (server/start-server {:port 9999
2                       :name "socket-repl"
3                       :accept 'clojure.server/repl})
4 ;; then e.g. telnet to localhost 9999
```

Socket REPL. This is how easy it is to start the socket REPL. clojure.server and then call start-server. Pass in a port and a name. So this is a name, so that if you start more than one of them you can keep track of the ones you have started. And then the accept function is the function that runs the server. clojure.server/repl is a REPL entry point. So this is just starting a REPL. And then you can telnet in. And you can do this anywhere.

Also, here is a super interesting thing. There is a Java system property, or a set of Java system properties, that allow you to specify these things, in which case the REPL will start automatically when Clojure loads. Which means that if you wanted to use a REPL to debug a Scala application, all you would have to do is have the Clojure JAR on the classpath, and start with the right flags, and it would now be a Scala application with a Clojure REPL running in it. And then you could just sort of tool around in there and do whatever you wanted to.

How many people have read "The Joy of Clojure"? Excellent book.

[Time 0:16:57]

slide title: REPL at Point of Interest

```
 1 (defmacro break []
 2   `(clojure.main/repl
 3     :prompt #(print "debug=> ")
 4     :read readr
 5     :eval (partial contextual-eval (local-context))))
 6
 7 (defn foo
 8   []
 9   (let [a 1]
10     (break)
11     a))
12
13 (foo)
```

https://github.com/joyofclojure/book-source/blob/b76ef15248dac88c7b1c77c2d461f3aa522a1461/first-edition/src/joy/breakpoint.clj

You may remember that they showed this little example of creating a REPL at a point of interest. So here is a macro called "break". "break" is a breakpoint. It is a debugger type thing, except because we have a reified environment, we do not need a separate debugger API, or anything special. All we need to do is launch a subREPL with one particular detail, which is: we want to know what the locals were.

So whenever you launch into the debugger, you can always see the locals in the frame. Well, a local context – you can go look at the code in "The Joy of Clojure", or on line – "local-context" is a little macro that pulls out the information about the locals. And then "contextual-eval" – notice I am overriding eval now when I start this REPL. I am using the regular read. I am overriding eval using this thing called contextual-eval. contextual-eval just evals with these additional symbols defined [as returned by local-context].

And so what happens here is, inside of my function I can say "(break)". I will now get a REPL, and in that REPL what is the value of "a"? Inside that REPL I have access to the locals. So it is 1. And I do not have to be limited by what GUI tools ship in my IDE debugger. I can stop, make a REPL, and then make Swing UI. I get it. Sometimes when you stop and you are debugging, you want a richer visualization than just text or just pretty printing. But you can make anything. Nobody is stopping you at this point.

[Time 0:18:26]

```
slide title: Launch UI From REPL

1 (-> *e
2     Throwable->map
3     clojure.inspector/inspect-tree)

[figure showing window created by inspect-tree call]
```

You can also launch UI from the REPL. So here at the REPL I am grabbing the most recent exception, using a very helpful little function which is little known: Throwable->map. Throwable to map will take in a Java Throwable and turn it into data. And Clojure uses that to print stack traces as data, but you can use it to make anything into data, and then as soon as you have done that you can turn around and pass it to any GUI viewer you want.

So here I am saying, "Oh, I just hit a problem." Rather than looking at a big stack trace at the REPL, I want to see it in a tree widget. Clojure has one. It is very trivial and ugly. clojure.inspector/inspect-tree.

Do the people in this room need for the gods of Clojure to write a better GUI tool in order to make this better? No. Who can make this better for themselves? Anybody. It takes a little bit of Swing fu. And by the way Seesaw, Clojure Seesaw, is a great little library for bringing up Swing controls. And there is a great REPL tutorial which shows, on the fly, at the REPL, just manufacturing little bits of UI when you need them.

So if you want to play this game, if you want to do the kinds of things I have been talking about, you need to know a few mechanical things.

[Time 0:19:46]

```
slide title: The Edges of the REPL

Initialization

Prompt

LineNumberingPushbackReader

Bindings

Exiting

  :repl/quit please!
```

`Use clojure.main/repl until you can't`

https://github.com/clojure/clojure/blob/master/src/clj/clojure/main.clj#L177-L269 [link to source code of clojure.main/repl]

You need to understand that there are a couple of rules about being a REPL that go beyond read, eval, print, and loop. There are not many. There are a few.

One of them is, you may want to do some initialization. And so when you look at clojure.main/repl, it allows you to override read, eval, and print. It does not allow you to override loop. So it always loops back unless you exit the REPL. But you might also want to override initialization. You might want to do something as part of starting a REPL. So that is another thing that you can do.

You can override the prompt. If you have only ever made one REPL, you probably don't care very much about the prompt. What is the default prompt in Clojure? It is the current namespace you are in, which starts as "user", followed by the equals character and the greater-than character. But you can make the prompt anything. And you may want to make the prompt something if you have more than one REPL. As soon as you are like, "I am going to run five REPLs against this program," then if they all have the same prompt, you may have trouble keeping track of which one is which. So that is another thing you want to run.

You also have to have a LineNumberingPushbackReader. So in general, a REPL is just about reading an arbitrary stream, which in Java would be a Reader. But if it is not a LineNumberingPushbackReader, then all of the things in Clojure that can tell you line numbers about problems would not be able to work at the REPL. So if you read clojure.main/repl, which is only like 30 to 40 lines of code, you will see that it provides all of these things. It does all of this for you, so you do not have to. So you can just call clojure.main/repl and make multiple different REPLs that suit you.

Another thing that the REPL does is, it sets up initial bindings of the dynamic vars. So there are a set of dynamic vars that Clojure programs assume will have bindings. I can never remember what the list of them is. But there they are in clojure.main. And so when you start your own REPL using clojure.main/repl you will get the bindings set up correctly as well. So it is really worth reading this one little function to understand – it is not very long – the guts of what go on in a REPL.

And then you want to have an idiom for exiting. And if you are writing your own REPL in the future, please make sure that you treat :repl/quit as a keyword – treat the evaluation of that as a trigger to exit. The original REPL, Clojure's REPL, did not do this. It used what? Control-D. Control-D is not always available, depending on how you are communicating with the REPL. If you are communicating over a socket, that is not going to work. So respect :repl/quit as well. And you can see that if you look at clojure.main/repl it does not respect :repl/quit, but clojure.server/repl – the one that got added for the socket server – respects :repl/quit. You should respect both of those things as ways of exiting the REPL so you can be compatible with others.

And the short piece of advice I would give here is that you do not actually have to make – I mean, clojure.main/repl is almost all hooks. If you look at it, it is like "call the initialization hook, set up the bindings, get into a loop, call the read hook, call the eval hook. If the evaluation is successful, call the print hook. If it is unsuccessful, call the throw hook. Check to see if you should exit, and go back." You could write that code yourself, but there are fiddly bits about using the LineNumberingPushbackReader and bindings, and so my advice to you is that if you need to start a subREPL, use clojure.main/repl or clojure.server/repl, until it irritates you. And when it irritates you, set yourself free from the training wheels and just write your own read, eval, print loop.

Audience member: Is the prompt going to be a function?

Yes. Uh, maybe not. That may be one of those things that will irritate you and cause you to want to write your own REPL. I will tell you the one that irritates me. The thrown handler that main provides takes as its argument the error that was thrown. If I was going to make a super awesome beginner environment for Clojure, I would want the thrown handler to take two arguments: the error that was thrown and the input

that caused that error, so I could do polymorphic dispatch on both of them and say whatever. It is dumb that clojure.main/repl does not do that, but it is like 15 lines of code. You just copy and paste it out and do your own thing.

And by the way, once you understand how the REPL works, do not write another function that takes a bunch of hooks. That is silly. Just write a function that does what you want to do. It is only a few lines of code anyway. Having a function that is all hooks is kind of weird.

[Time 0:24:24]

```
slide title: Dynamic Development Examples

cloxp

KLIPSE

Light Table

Seesaw REPL tutorial
```

- cloxp - https://github.com/cloxp/cloxp-install
- KLIPSE - http://blog.klipse.tech/clojure/2016/03/17/klipse.html
- Light Table - http://lighttable.com
- Seesaw REPL tutorial - https://gist.github.com/daveray/1441520

So to give you a couple of examples of things that you could play with to see the REPL. These are all links. I will drop the slides and drop a Twitter link to these, so you can grab these links and look at them later.

cloxp is a Smalltalk-like environment.

KLIPSE is a multi language execute in the browser thing. So if you are writing documentation you can actually put forms in your documentation, and people can tweak them and eval them on your documentation page and see what they do. So it is kind of cool for exploration.

Light Table obviously is a full on IDE, written in ClojureScript.

And then, those things are all kind of fancy and graphicable-ble-ble. The Seesaw REPL tutorial is a beautiful example of the kind of thing I am talking about with building up your programs at the REPL. Dave Ray just goes through and says, "Here I am sitting at the REPL. I am the author of Seesaw. I want to make some Swing stuff. Let me try this. Let me try that. Let me try the other thing." And build up programs from that.

So now we have talked about the "How?" part. We have talked about the "Why?" part, and we have talked a little bit about the "How?" part. The mechanics of what it is like to manipulate a REPL.

[Time 0:25:40]

```
slide title: REPL-Driven Development

Make (revisit!) a plan

Aim small, miss small

Save everything

Promote proven things (accretion, relaxation, fixation)

Visualize
```

And now we are going to get to the super interesting part, potentially, which is: what are you going to do with it? And so I am going to give several different ideas of what I think contribute to REPL-driven development. And I will skip this first bullet for a second.

"Aim small, miss small." Does anybody know the origin of that quote? I do not think that I know the original origin of that quote, but the place I have seen it is in what I think is quite an underrated movie called "Braveheart Carolina". So "Braveheart Carolina" is about Mel Gibson defending the Carolinas from the British during the Revolutionary War. It actually was not called "Braveheart Carolina". It was called "The Patriot". But I think it is a really good movie. If you like fake historical action movies, it is a really good movie.

But there is a scene where his sons have to shoot the bad guys, and one of them looks to the other and says, "Remember what Dad says: aim small, miss small".

And what I mean here is: do small things, and make small mistakes before you move on. And you can absolutely look like a genius in the Clojure world by just sticking to that rule. And I started keeping score on this. I would say that half of the time when I go out and look on the Internet for a random beginner struggling with Clojure, and they have posted a question on StackOverflow, or they have asked a question on the mailing list, 90% of the time I can split their problem into smaller pieces, completely naively – like put the blinders on. I see that your form has four sub-forms in it. Evaluate each one of those four sub-forms at the REPL. Figure out which one of them is the problem. And then go back to them and say, "Hey, this is the one that was the problem." And they are like, "Oh, my god. You are so smart!" You have been doing Clojure for ten years. I reply "No, this is a really dumb tactic." This is just: take a big thing that did not work, and break it into small things and see which one of them did not work.

And of course the implication on the other side of it is: why not just make the small things first to begin with, and then try them out before you put them in the big thing?

Save everything. Your interactions with the REPL – I am baffled when people type things into the REPL. We will talk about that a bit more in a second.

And then you promote proven things. So you work at the REPL, and then when you get something that you like, you put it in a library. And you do that with the Clojure idiomatic way of not changing the semantics of names. So if you published a name out to the world and you said it does this, then you do not change the rules of how that name works. You make a new name when you want to do that.

[Time 0:28:17]

```
slide title: Make a Plan

Fast feedback loops support incremental development

  Incrementalism can only find a local maximum

  Step back and know your objective

"Not all those who wander are lost"

  Just most


Debugging Too: https://www.youtube.com/watch?v=FihU5JxmnBg
```

Debugging Too: https://www.youtube.com/watch?v=FihU5JxmnBg

[YouTube link is to Stuart Halloway's talk "Debugging with the Scientific Method"]

But now let us go back to the first point, which is: make a plan.

I do not actually believe in REPL-driven development. I do not think the REPL can drive. Just like I do not think tests can drive. I think *you* have to drive. It is really passionate capable human driven development, or something, but it does not make as good an acronym, and it takes too long to say.

So the idea here is that the REPL gives you a really fast feedback loop, faster than, say, TDD. But the REPL, and TDD, and any other incremental feedback loop process, is subject to the limitation that it can only find a local maximum. If you start here, and you take little small steps from here, you are never going to find the solution that is over there. And that is on a person. And in order to do that you have to step back and know your objective.

And the way I summarize this, borrowing from Tolkien, is that not all those who wander are lost, but most are. So you cannot just wander around. And this *is* a problem, because both TDD, and Clojure even worse, when you are working at the REPL like I am describing, encourage getting into a fugue state, where you are just sitting there like it is so short term rewarding as a developer. It is like, "Oh, I will try this, I will try that, . . . " And you lose track of the idea that you had an objective to begin with, and you go down the rabbit hole. Or I guess the more developer idiomatic, you shave the yak. You were working on this, and all of a sudden you realize it is really easy to wander off at the REPL. So don't.

[Time 0:29:50]

slide title: Aim Small, Miss Small

```
 1 (defn foo
 2   [n]
 3   (cond (> n 40) [1]  (+ n 20) [2]
 4         (> n 20) [3]  (- (first n) 20) [4]
 5         :else [5] 0 [6] ))
 6
 7 (defn n 24)
 8
 9 ;; results evaluating with cursor at each position
10 [1] => false
11 [2] => 44
12 [3] => true
13 [4] => Broken! HaHa!
14 [5] => :else
15 [6] => 0
```

http://blog.cognitect.com/blog/2017/6/5/repl-debugging-no-stacktrace-required

So I did a blog post about this last week about this sort of "aim small, miss small" idea. And somebody else in the Clojure community had done a blog post on debugging and tracing tools, and showing how to debug and use various traditional developer tools to find a problem.

And their example function was this function "foo", which takes "n" and then goes down a cond, which has three branches and three things that it does in those branches. And this function foo, when you call it with the number 24, fails. And what the blog post walks through is how I would approach solving this at the REPL if I had no idea what was wrong.

And that is, I would set up a local context in which each of those sub-forms, [1], [2], [3], [4], [5], and [6], could be evaluated, and then I would evaluate them. I would not type anything to do that, because any Clojure editor worthy of its name will let you put the cursor near a form and then do what? Send it to the REPL. Don't type into the REPL. When I watch somebody type into the REPL, it is like dragging fingernails down a

chalkboard. Type into a nice buffer in your favorite editor, where all of your nice comfortable tools are, and then send things to the REPL. And then when you are done, you have this buffer that had all of the history of what you did, and you can save it and hold on to it.

And in this case I evaluate this form [1], this form [2], ... and this form [6], and I discover that the fourth evaluation is the one that is broken. And this is a very powerful technique. In the blog post I actually hooked the REPL to hide all of the error information. So when the exception is thrown, instead of giving you the stack trace and all of the useful juicy details, it just prints out "Broken! HaHa!". So you actually have even less information. And with even less information the problem is still trivial to solve, if you break it apart into parts.

So I really hope that in a year, I can say this to an audience, and they can all say, "OK, move on." Right. Everybody in the whole ecosystem gets this and we do not have to talk about this any more. It does not take any cleverness. This is an O of N search through the problem. I am just going step by step. It is not smart, but it will find the problem a lot faster than going off, and what we see people doing

[Time 0:32:17]

[switches back to earlier "Make a Plan" slide]

is going off without a plan.

I should mention that – and people may not know this – that one of my responsibilities in my day job is that I manage technical support for Datomic and Clojure. So when people have a commercial support relationship with Cognitect and they have questions, then I do not necessarily answer all of the questions, but I lead the team that does that. And so that has given me the opportunity to be exposed to a lot of other people's pain points. And the really bad ones. The ones where somebody would say, "This was bad enough that I called for help and spent money to get help on the problem."

And it is amazing how many of those pain points – they almost always start with a problem, and sometimes a subtle one. Usually by the time we get a call, the problem has been made worse. And the problem has been made worse by somebody going off without a plan and just doing stuff. Going off without a plan is bad, and I mean doing stuff in the imperative sense of the word. Like, "I changed the system," and not in our happy Clojure immutable sort of way.

So I am very passionate about these things, because I have seen people really hurt themselves. I mean really take a situation that was bad and make it actively much worse, by not having a plan, and by not aiming small and missing small, by going off and trying some big thing and making a mess.

[Time 0:33:46]

```
slide title: Save Everything

Don't type into a REPL

  Type into a buffer

Save into a file

  No organization required
```

The next point is: save everything.

When I am developing, I type into a file. For a long time – emulating what I believe Rich does – no one has actually ever seen Rich type, so there is a lot of theorizing and hypothesizing about what he actually does. Sometimes I think he just stares at the screen and the Clojure code appears.

But what I did for a while, and what I think he does, is I had one file that was eventually several tens of thousands of lines long, that just had every line of Clojure code – experimental, you know, whatever I had ever typed. Which is fantastic, right, because then you can say, "What was I thinking about the other day when I was doing X, Y, or Z?" And it is just right there in the file. And of course text search and structural search are powerful enough in our tools now that you cannot, as a human, write enough code in the rest of your life that it would be hard for you to search all of it, if you had it in one place.

For no particular reason, I now create a new file every month. And they are all in the same directory, so right now I am working on the June 2017 file. But it just has all of the Clojure code that I thought about in June 2017. And of course the stuff that is actually going to go into Clojure, or into a library, or into production code, also gets pulled out of there and put into wherever it is going to go.

An example of how this helped me recently was: we were bitten by a classloader related problem where – well it was complicated. It does not really matter. But we were bitten by this problem that caused us to walk away from including some JMX based tools into Datomic that would improve the way we log. And a year and a half later I went on the Clojure bug list and talked about how to avoid this classloader problem, and then I fixed our build that we use internally to avoid this sort of problem. And then I said, "You know, a year and a half and I almost had this working." This thing that was going to build on top of this thing, which at the time was broken, I just went back and searched through my files, and found that code and pasted it in, and ten minutes later I had a new feature that I did not think I was going to have. Because I just remembered: oh, yeah, you know what? I had done that.

And of course if I had typed everything into the REPL, or threw away this stuff, it would just be gone. I cannot remember what I did a year ago, in its totality. I can barely even remember even at the pointer level. Even when I know that I did something a year ago, I can't think of a good enough name to search for it to find it, sometimes, and then I have to read through a bunch of code. But do not throw stuff away.

Audience member: What is your signal to noise ratio? How often do you find good stuff?

Oh, you mean in the stuff I saved?

Audience member: Yeah.

So good stuff in terms of poking and prodding at programs: all the time. Good stuff in terms of things that are going to actually become production code, almost none of it.

But a lot of it is like expressions that you can evaluate against a piece of production code and do something interesting with it. So it is a library of expressions, really.

[Time 0:37:00]

```
slide title: Promote Proven Things

REPL "branch coverage"

  usually by building up from leaf notes

Evolving

  Accrete (provide more)

  Relax (require less)

  Fix
```

And then: promote proven things.

So take things that you have developed from small pieces at the REPL, and put them wherever they go. Put them in a namespace. Make them part of your project. Add tests for them to your test suite, and so forth.

And when I am developing, I do not write unit tests for everything. But I have what I would call 100% branch coverage, in terms of: I have watched at the REPL every branch of my code execute, before I put it anywhere.

I was talking to Ben earlier today, and he was talking about that he always has test level coverage of all of those things. Which is fine, too. I am certainly not saying you cannot do that.

But I start from the small pieces. This ["leaf notes"] should say "leaf nodes". And in fact, this did say leaf nodes, and Mac has done this thing in Keynote now where they auto save for you. They are trying to take "save" out of your control and make it impossible for you to delete something by accident, and they caused me to move back to a different version of this than I wanted, by not giving me control over when I save things. I am not a big fan of this change. Anybody else has been burned by that? All the Apple stuff that does not want to let you ... Also, it wants to put everything in your iCloud account.

So I was recorded for the defn podcast. I do not know if anyone has listened to it. It was recorded last weekend. And the way they do their recordings is, everybody does their own recording locally, and then they sync it up, because that way you get better audio quality. So they take each person's audio track and sync it up.

It was actually quite amusing, because in order to provide a sync point, everybody has to clap. So we are sitting there on video together: all right guys, on three. Is it on three? Or is it after three? And when we did it, it was a total fail, because there was video lag. So Vijay would say, "All right guys" and he would freeze. And Ryan and I would be like, "Uh" [clap]. It was a total mess. But the audio sounds right at the end. It sounded OK. So it worked out.

And now I have completely lost track of what I was talking about. Where was I?

Branch coverage. Yes! Thank you. I should look at my slide. See, this is the memory problems I was talking about.

So I build up from leaf nodes. I have seen the branches work. And if I am really worried to having the branches work, I do not know if it really works, I am probably going to write a generative test. I am probably going to do something more than push examples through it. I am going to write a generative test using spec or something like that.

And then these rules for evolving. They are really not about REPL development, but this is the talk that Rich gave at Conj last year, Spec-ulation, about being kind to consumers of your libraries. Where kindness is about not making their programs stop working.

And the way you make programs continue to work is: your functions can provide more. So a function that used to give you "A" can give you "A and B". That is not going to break your program, because you were looking for "A".

You can relax. You can require less. Programs that used to take a map with three keys in it, can now take a map with two keys in it, and if you continue [as a user of the library] to provide three keys, that is OK, too.

And then obviously we fix bugs, but we do not change the meanings of names. And if we do this as an ecosystem, we will be a popular and powerful language 20 years from now, because nobody else seems to get this. Very few other people seem to get this. Java gets this, and people hate them for it. But Java, they work very hard not to make breaking changes to the meaning of names, at the cost of leaving around stupid names, and marking things as deprecated. One of our coworkers at Cognitect used to call it "depreciated", because it was not actually going away, it was just sort of sucking out value.

But does everybody understand what I am talking about here? If you have shipped a library out into the world, and you do not like the name, tough tootie. You shipped. And if you so desperately dislike it, then make a new namespace. If you have fred/wilma, and you do not like the meaning of wilma, then make fred2/wilma,

or fred/wilma2. It is OK to do that. You will find you do not need to do that very often. It is not like you are going to end up with wilma74, or something like that. But even if you did, that is better than breaking programs.

Audience member: Is this why people always ask why there are so many old libraries in Clojure? Like: "It hasn't had a commit for 7 years?"

Well, it is part of that.

Different audience member: Maybe it has not had a commit for 7 years because it still works.

That is the thing. I feel like if I answered things on Twitter with that answer, people would see that as trying to poke a fight, but I am tempted. I am tempted to say, when somebody points at a library that I wrote, and say, "I see that this has only had 3 commits in the last 5 years." I am like, "Wow, that is great. I only had 3 bugs, or 3 things worth making better."

That is not always true. Things are abandoned, too. So it is easy to find examples of things that were never good, or that people walked away from. So I am not going to claim that is always true. But it is sometimes true.

[Time 0:42:25]

```
slide title: Visualize

(this space intentionally blank)
```

And then: visualize. This space is blank. We need better tools. We need better data visualization tools. And we need them to be a la carte and available from the REPL. I am willing to use frameworky things if I have to. I am willing to use IntelliJ IDEA. It has got great stuff in it.

But I would much rather have the ability to manipulate and visualize data delivered in a plain form that is not dependent on running inside of Eclipse. That is not dependent on running inside of IntelliJ IDEA. I wish that Gorilla REPL could just be started on an arbitrary program, instead of having to be started as its own thing. It is cool! Gorilla REPL has a lot of the kinds of ideas and things that I am talking about, but I do not want to have to get into it. I want it to be available to me as a library.

So I think we have a ton of work that we could do here. And I suspect we could do really amazing stuff using Seesaw, in short order. We already have a pretty amazing library for manipulating Swing. I get it. Swing is not cool, or whatever. But I am talking about tools for developers. And remember that IntelliJ IDEA is built in Swing. So it is possible, with sufficient elbow grease, to make it beautiful.

[Time 0:43:25]

```
slide title: Calls to Action

Live at the REPL

Make Java better

spec something

Improve the docs
```

So I would give you these calls to action, how to take the ideas from tonight and do something with them that will hopefully make your life, and the Clojure ecosystem's life, better.

One is live at the REPL. So some specific things you can try at the REPL.

Two is make Java better. Remember, you are the lucky few. You have been chosen, and 9 million people are going to write assembly language code, and you are going to come along and stitch it together and build applications, and take all the credit. But we need to add value over just building things out of Java, and that means taking Java libraries and doing better things with them.

Write specs. Specs are new. We don't have specs for everything yet.

And improve docs.

[Time 0:44:10]

```
slide title: Live at the REPL

Try the techniques shown here

Hook a REPL to provide assistance with a common problem

  enumerate the powers at your disposal

Make a custom inspector
```

So let us talk about each of these. Live at the REPL.

So here are a couple of specific ideas. One, go back through this slide deck and just try the various things that I have talked about, if you have not done it before. Try starting a REPL with a custom prompt. Try starting a REPL with a custom printer, or a custom thrown handler.

Then, slightly more advanced, find a common problem that people have, and think about how you would use the REPL to say, "Oh, I see you are having that problem. Here is some extra help."

And, as an interesting sub-bullet to that, enumerate the kinds of powers you have at your disposal if you wanted to do that. If you are a function written in Clojure, with access to all of the Java libraries in the ecosystem, and you have in hand a data structure that was created by a Clojure programmer, and an exception that came from trying to evaluate that data structure, what kinds of things could you do to make things better? To help that person when they encounter that situation.

And then finally, make a custom inspector. Use Seesaw, or something, and if you are in ClojureScript use a GUI tool that runs in the browser, or whatever, and make a custom data inspector.

All right, get to it!

No, we do not have to do it right now.

[Time 0:45:23]

```
slide:

        Clojure:

The Language for Expressivity
      on the JVM
```

And then let us talk about making Java better.

So I think Clojure should aspire to say with a straight face that we are the language for expressivity on the JVM. And the comparison that I want to make here specifically is to Python.

[Time 0:45:40]

```
slide title: Data Science

"Every time I need
to quickly prototype
something that just
works, I end up
using Python."

[cover of book "Data Science from Scratch"]

Grus, Joel (2015-04-14).  Data Science from Scratch: First Principles
with Python
```

This is kind of a cool book, Data Science from Scratch, that is about doing data science in Python. And where the author is explaining the choice of Python for the book, they said, "Every time I need to quickly prototype something, I end up using Python." And they specifically said, "I do not necessarily think Python is the best choice for any particular thing. I am not trying to sell it here, except that I just keep coming back to it."

And so it is worth asking:

[Time 0:46:08]

```
slide title: Why We Love Python

Python Can                    Clojure Can

interactive (shell)           interactive (REPL!)

reach C-linkage algorithms    reach Java-linkage algorithms

lines and strings             edn

comprehensions                transducers

                              spec
```

Why do we love Python?

Well, if you are someone who wants to consume C-linkage libraries, which is what people use Python to do – the actual algorithmic fast stuff is not written in Python. It is written in C-linkage libraries – why would you want to use Python?

Well first, it has this great interactive environment, the shell. Well does Clojure have one of those? Uh, yeah. And we think REPLs are better than shells.

When you are using Python, you can reach – because other people have already done the work to write the wrappers – you can reach C linkage algorithms. With Clojure, can we reach Java linkage algorithms? Yeah, and in fact it is easier in Clojure. In line Java in Clojure is easier than extending Python to reach C linkage things. So I think advantage, again, Clojure.

Python manipulates lines and strings. It is a textually oriented language. Clojure has edn. Advantage? Clojure again.

Python has a powerful mechanism for working with data structures called comprehensions. Clojure has an even more powerful one called transducers.

And then Clojure has this new thing, spec, which very few other languages offer anything that matches.

So we are well positioned. So why is this not true already?

Audience member: library support?

Library support. Documentation. Time. Python is substantially older than Clojure. There are lots of reasons why it is not true. So there is work to be done.

Another reason is that C linkage libraries – so first off, I am going to stipulate that reaching C linkage or Java linkage is a super advantageous thing to do, and I am not going to say which one is better. I am just going to say they are both really good. And there is no point in Clojure trying to compete in being the language of expressivity for reaching C language, because it would be a much harder battle, because we would have to fight against Python, which is already doing a good job of doing that. Whereas in the JVM ecosystem we do not have, in my opinion, other good choices.

But one of the nice things about C linkage, is a lot of those C linkage libraries were written before our minds got poisoned by object orientation. So a lot of the Java linkage libraries are unnecessarily object-y.

[Time 0:48:24]

```
slide title: Undo the Damage of BGFS

Builders, Getters, Fluent APIs, and Setters (BGFS)

  pronounced "big gaffes"

  necessitate programs where information would suffice

  turn a single moment in time into unnecessary stateful interactions

  unnecessarily specific
```

So one of the things we can do as Clojure programmers is: undo the damage of this.

And one example of this damage is the use of builders, and getters, and fluent APIs, and setters. BGFS, which is pronounced “big gaffes”.

And the thing is, when you build programs that are filled with builders and getters and all these APIs, they turn what should have been data into a series of stateful interactions. I want to make a chart. Create the chart object. Modify it this way. Add this thing to it. Add this thing to it. Add this thing to it. Do I want a chart that is halfway made with some things added to it, and not others? I do not *ever* want that. I never want that.

So builders, getters, fluent APIs, setters, these are just bad news. They are bad news for programming.

[Time 0:49:13]

```
slide title: BGFS

 7
 8    double[] yData = new double[] { 2.0, 1.0, 0.0 };         unnecessary
 9                                                              state
10    // Create Chart                                             |
11    XYChart chart = new XYChart(500, 400);                      |
12    chart.setTitle("Sample Chart");                             |
13    chart.setXAxisTitle("X");   <-- specificity                 |
```

```
14    chart.setXAxisTitle("Y");                                    V
15    XYSeries series = chart.addSeries("y(x)", null, yData);
16    series.setMarker(SeriesMarkers.CIRCLE);

                ^
                |
                |
              program!
          (could have been data)
```

https://knowm.org/open-source/xchart/xchart-example-code/

And happily, we can take this and make it better. So here is a nice little Java library called XChart.

An so XChart is like: well, I'm going to make a chart. I am going to set the title. Setter, setter, setter. Fluent API [probably referring to line 15], fluent API [line 16?]. And this could have all been data. I could have had data. If I had data, I could hand it to somebody else. And they could have written a program that consumed it separately from the original program. But now I have a program that I have to run.

It is unnecessarily specific. It has all of these specific classes, instead of generic data structures. And it has unnecessary state. Every one of these interactions from here [line 11?] until I am done are things that I would not ever want the outside world to see. There is an unnecessary time line that has been added to the program.

And you can fix this. And this now has an official name, as of today.

[Time 0:50:03]

```
slide title: Datafication
```

```
1 (c/view
2  (c/pie-chart    <------- functional constructor
3   [["Not Pacman" 1/4]
4    ["Pacman" 3/4]]
5   {:start-angle 225.0         <----- generic data
6    :plot {:background-color :black}
7    :series [{:color :black} {:color :yellow}]}))
```

https://hypirion.github.io/clj-xchart/examples

It is called datafication. Datafication is when you take a program that is guilty of big gaffes. It is full of builders and getters and setters. And you write a wrapper for it in a language like Clojure, and turn it into a functional API. So this is a library written by Hypirion – I do not know who that Github handle actually belongs to as a human name – but this is a library that wraps XChart and turns it into something that is purely functional.

So notice that now we have a constructor pie-chart that just takes a bunch of data. Which means that if I want to generate charts, how can I assemble this data? Well I have the Clojure sequence API, I have transducers. Every single function in Clojure core that deals with data helps me assemble this.

How many functions in the core of Java help me do any of this? [switched back briefly to previous slide with Java example] None. This is an API that got made up for XChart.

So this is an area where we can add value. I picked this one as a particular example because someone has already done the work, but you can go and look at it and say, oh, I could go and do that for a Java library.

[Time 0:51:12]

```
slide title: spec Something
```

```
[figure that looks like examples of XChart library charts]
```

https://github.com/hyPiRion/clj-xchart

The next thing is you can spec something. So I spec'd clj-xchart, which is the wrapper on XChart. So the last two slides had this chart library. And these are random charts that I generated. So once I had spec'd it – maybe you haven't seen spec yet, but one of the things you can do with spec is generate example data. So having spec'd clj-xchart, I then just started generating random charts and looking at them to sort of see what I had.

[Time 0:51:41]

```
slide title: Why spec?

More expressive than Java types

  or less expressive, if you prefer -- you pay as you go

Better language integration

  generate data and tests

  generic data enables better tooling
```

And why would I want to do this? Well, spec is more expressive than Java types. There are a ton of things you can say about the structure of data in spec, that cannot be said, in an easy way, with a Java type.

It is also less expressive, because you do not have to use it. It is expressive, but it is also opt in. So unlike a type system in something like Java, with spec you can say, "I will use it on the stuff I care about." I chose to actually care about everything in this library. I did spec it reasonably fully, but you do not have to.

spec has better language integration than Java types. It is integrated not just at compile time, but at test time and at run time. You can make tooling that you are going to use at other times in the work flow.

And because it is made out of data, other people can do cool stuff with it.

[Time 0:52:24]

```
slide title: clj-xchart xy-series specviz

[figure of something]
```

https://github.com/jebberjeb/specviz

So this is a little library called specviz. So this is Jeb's library. Jeb Beach, who works with David and me. So he wrote this little library that takes specs and then runs them through Graphviz, and makes a visualization of the spec.

So this is everything about XChart. And obviously you cannot read it at this resolution, but I will tell you that this is a much easier way to see the whole surface area of the API. Because this is the whole API. It is everything you can do.

These gray boxes over here are the functional entry points. Because they are pure functions, there is only one for each different chart type. And then these are all the different argument types. So you could look at this while programming and sort of have the whole thing in your head.

[Time 0:53:13]

```
slide title: Example-Based Generators

(def generators
  {:com.hypirion.clj-xchart.specs.chart/title
   #(s/gen (into #{} (map (fn [n] (str "Example Title " n))) (range 1 5)))
   ::series/chartable-number #(s/gen double?)
   ::series/series-name #(s/gen example-series-names)
   ::series/xy-series-elem #(->> (s/gen ::xy-series-elem* generators)
                                 (gen/fmap force-axis-counts))
   ::series/show-in-legend? #(s/gen boolean?)
   ::sty/width #(s/gen #{200 300 400 500})
   ::sty/height #(s/gen #{100 200 300})
   ::sty/color #(s/gen ::sty/builtin-color)
   ::sty/marker #(s/gen ::sty/builtin-marker)
   ::sty/stroke #(s/gen ::sty/builtin-stroke)
   ::sty/font #(s/gen ::sty/builtin-font-set)
   ::sty/theme #(s/gen ::sty/builtin-theme)})
```

You can also help with generating example data. So when I was doing the specification exercise for XChart, there were several things that I did not want to generate from the entire range. So if you are generating data from the entire range, it is going to do things like: I am going to make a chart whose size in the X dimension is 1 pixel and whose size in the Y dimension is a trillion pixels. If I am testing to make sure that the Swing GUI does not blow up on weird inputs, then I might want that, but I do not actually think that anyone is going to make charts like that.

So one of the things you can do with spec is you can provide – it is this easy to provide custom generators. And by this easy, I mean a map from names of things to sets. So notice that most of these are sets. So I am saying whenever I need a chart width, it is going to be 200, 300, 400, or 500. When I need a height, it is just going to be 100, 200, or 300.

So this provides a lot of value as a really easy way to generate useful domain data.

[Time 0:54:17]

```
slide title: "Discovered" Specs

(defmethod data-compatible-with-render-style? [:xy :area]
  [series]
  (ordered? (:x series)))


(defn finite?
  [x]
  (not (or (Double/isInfinite x) (Double/isNaN x))))


(defn axis-counts-match?
  [{:keys [x y error-bars]}]
  (and (= (count x) (count y) (count (or error-bars x)))))
```

And you can discover information that was not present in the Java types. So when I learned how to use XChart, one of the things I discovered was that if you make an XYChart with the area render style, then the X data series has to be ordered. Guess what happens in XChart if that is not true. It perfectly happily makes a data structure that represents the chart, and then when you try to render it, it throws an exception on the

AWT rendering thread.

So by capturing this information as a predicate – and similarly for finite. It turns out that the chart library does not want to try to render infinite. I mean they could have just left those values out, but it throws an exception if you try to render infinite, again on the AWT rendering thread.

The axis counts have to match. So you have to have as much X data as you have Y data. They did not have to do it that way. They could have said if you have 7 X and 5 Y, we will just show 5 and we will drop the last 2 X, but whatever. They made those decisions.

These discovered specs add additional knowledge about how to use the system that the person who wrote the Java code clearly knew. They made all of these decisions. But the decisions are not made visible in an easy way.

I will tell you one more story about these specs. I spec'd clj-xchart, which is the Clojure library, which wraps XChart, which is the Java library. When I started generating example data and rendering it on the screen, within 5 minutes I had uncovered a JVM bug. Not a clj-xchart bug, not an XChart bug, a JVM bug that crashes the JVM on the Mac and on Linux. And I do not know what Windows does. Maybe it does it there. Maybe it does not. I do not have to test it.

But it is a compelling demonstration of the benefit of spec, and the benefit of generative testing, that I was able to find it. I mean, how often do you find a crashing JVM bug? Hopefully not very often, right?

Audience member: What is the bug?

I don't know. So it is interesting. Gary and I were talking about this earlier. So Gary is currently maintaining test.check, which is the library that Clojure spec uses underneath to generate test data. And one of the things it will do is it will shrink the problem case. So when it finds a great big ugly piece of data that causes your program to break, it will shrink that data step by step, until it finds a more tractable example.

Of course the problem that I had is that the way you normally use that, is it runs in your process. So inside of your process it makes some data, it sees the failure, and then it makes other data. Well, in my case, when it failed, it kills the process. So in order to let spec shrink it, I would have to write something that generates the data, shells out to another Java process, uses the data over there, checks to see if it crashes or not, and then comes back. I mean, it is still worth doing. I just have not done it yet.

Audience member: Use a REPL!

Well you could definitely use a REPL. REPLs are good for everything. That is the new official Kool Aid. REPLs solve all the problems.

I mean, it happens to be passing a really large integer value to some API that expects only modest size integer values. So I suspect that something is being turned into pointer math at the actual native level, and not being value checked. So a bit creepy.

[Time 0:57:52]

```
slide title: Write a Guide

Explore at the REPL

Spec to taste

Help a Java programmer

[screen shot of the top of page https://clojure.org/guides/spec ]
```

https://clojure.org/guides/spec

The last thing I want to mention is: write a guide. I think that the reason that Clojure is not the language for data science on the JVM, like Python is for C linkage, is Python has great docs. We could have a lot better docs. We could have a lot more focused docs on particular tasks. Like hey, you are doing data science using this Java library. Here is how to manipulate it from the REPL. That kind of thing.

And I am happy to report, my slide here shows you there is this guide page on clojure.org. And since I made this slide, which is only like a week and a half ago, there is a new guide page here. So someone has done it. Not by my exhorting them to do it through this slide. It happened to already be happening. Has anybody seen the new guide?

Audience member: Weird characters.

Weird characters. We now have the weird character guide. It is just a guide to every punctuation character used in Clojure, indexed by the punctuation characters. Because they are hard to Google. It is very hard to Google for <<!. What does that mean? Because Google strips all that stuff out for the searching.

But think about writing documentation. Think about encouraging somebody who you have seen write good documentation in a blog to contribute it to clojure.org. The process is – the vetting is reasonably tough. We want the documentation to be of high quality. But it is done via pull request, and Alex Miller acts as the editor, and helps people shepherd things through.

[Time 0:59:20]

`slide:`

`cognitect.com`

So that is a set of thoughts about REPL driven development brought to you by Cognitect, my company, which flew me up here to evangelize Clojure this week. I was at the Java user group last night, and here tonight, and visiting a couple of companies that use Clojure and talked to them. So I really appreciate you all having me and giving me the opportunity to be here. And what I would like to do is open it up for a few minutes of questions. And then after that, for people who are interested, we will go somewhere more relaxing and dimly lit, with a broader selection of beverages.

Audience member: How does any of this change when you are debugging or developing a distributed application?

How does any of this change when you are debugging or developing a distributed application?

Audience member: Specifically around the launch GUI inspectors thing?

So you want to get the individual pieces working where you can. So if you have a distributed system that has a failure mode, you want to figure out which node, or which kind of node, is causing that failure. So I would say obviously the problem is harder with a distributed system, but I still think that all of these tools are relevant. It is just that the problem is harder.

I would say that the thought process becomes all the more important. It is a little bit like the example I was making a minute ago about shrinking. Let us imagine that some of these techniques are going to work 100 times slower in some special Martian environment. Well that is kind of distributed systems, right? So some of the things you are going to try to do are not going to be as easy. They are going to involve shipping data to where your console is, or whatever.

So the constant factors in the sort of process of finding the problem are going to be multiplied by 100. But anyone who has ever done algorithmic analysis knows: do the constant factors matter? No. The factor that you want to have, if you are tracking down a problem, is that you want the heuristic – I will not call it an algorithm, because we do not know how to solve problems – but you want the heuristic to have what performance characteristic? Realistically, in big O notation.

You want it to be 1. You want it to be big O of 1. There is a problem, and you put your hand up to your head and say, oh yeah, there is the answer. That is not going to happen, but what is the realistic thing that we can hope for? Logarithmic. It is realistic to do logarithmic problem solving, because that just means divide and conquer. That means saying I know there is a problem somewhere out here. What question can I ask that divides the world in half?

If you are good at playing 20 questions. I love watching my kids, especially when they were younger, and they were just terrible at 20 questions. And you would be like: I am thinking of an animal. And they would say: is it a really large elephant?

No. That is not how you do it. That is not going to be a logarithmic game. You want to make a logarithmic game out of it. So I still think that these tools are valuable. And I would also mention that data orientation and immutability are huge in distributed system. Data orientation means that if you find yourself in a situation where you need to move data from one node to another, in order to understand what is going on, because the node where the problem is happening is in AWS, and you cannot touch it very easily, or whatever. Then moving data around, if everything is made out of data, is easier.

And immutability matters even more, because when things go wrong, if you can walk up to a process and say: nothing ever changed here. We only added new things here. So whatever was wrong 10 minutes ago that caused the problem is still sitting there in memory. That is incredibly important for being able to reason about problems.

So the problem certainly gets harder when the system is distributed, but I think that these techniques, in varying ways, are still appropriate.

[Time 1:03:17]

Audience member: What sort of things do you think that are missing from the JVM that would actually make REPLs more powerful? So in some Lisps there are things like restartable exceptions.

Audience member: Something like a restartable exception would be really awesome, because that function that errored out, make it return X instead of Y.

So the observation is that traditional Lisps had continuation systems, so when an exception happened, instead of just having an exception, you have a new moment in your REPL environment that is like, oh, an exception has happened. Would you like to replace the calculation that was going with this value and continue? And Clojure does not have that in a first class way, because the JVM does not have it in a first class way.

Now Clojure users have written several libraries that do this with errors. So is anybody using any of the condition systems in Clojure? And I did a bunch of exploration about adding this to Clojure itself. So there was a point several years ago where I spent a couple of hundred hours working on what it would look like to make a condition system a more first class thing in Clojure. And all I ever got out of it was ex-info. So if you have used ex-info and ex-data, the only thing that we ever did after thinking about it really hard and long was we added datafied exceptions to the language, which is hardly what we started out to do. It is much less ambitious than that.

Which is just to give you sort of an idea of the long road that ideas take towards getting into the core.

You know, a condition system would be nice. There is no doubt. I am not – I am resigned to the platforms we run on, so I honestly do not spend a lot of time thinking about what could be. And this is a thing that is important when you are doing evangelism about Clojure, because one of the biggest complaints about Clojure from sort of language nerds is: wow, this is not nearly as pure Lisp as I want it to be. It really feels tarnished by the fact that it runs on the JVM. It feels tarnished by the fact that it runs on top of JavaScript. And it is like: no. We are all here getting paid because it runs on top of the JVM and it runs on top of JavaScript. So I do not bite that hand that feeds us. And honestly, I do not spend much time thinking about how much better could my life be if I had these kinds of tools.

One thing I would say is that having VMs that were friendlier to immutability – so immutability, if you knew about it at the JVM level, if you knew when you created a piece of memory, I am never going to change this, you have to imagine would facilitate optimizations.

Audience member: At least in the garbage collector

Yeah. Exactly, right? So I think that would be powerful, and it would not affect us as programmers, but it would maybe categorically change suitability of certain algorithms, or certain things that we might want to do.

And then one year I came up to Chicago and gave a talk about Clojure, and Dave Thomas was in the talk, and he sat down with me after, and he just sort of patted me on the head and said: this is all really cute, but everything should be done with these vector languages and advanced VMs, and whatever. And I am like, yeah that is really cool. When that spaceship lands, and people start using that in a widespread way, then we will build Clojure against that. But there are people looking at high performance platforms of the future. So I think performance is definitely a thing.

Another thing – you have got me started, now I am going through my memory. Another one would be, what do they call them, fixed numbers. The ability to have something that is unified with objects, but is a fixed numeric representation, because then you do not have the performance complications, and the complexity, around boxing and unboxing.

Audience member: I thought that was something that was planned.

[Stopped transcript at 1 hour 8 minutes in, during middle of Q&A with audience. Will listen to it and transcribe some more if it goes back to something I am interested in transcribing.]

[Time 1:07:47]

[Time 1:08:17]

tail recursion

Clojure libraries to take advantage of Hadoop and Spark without rewriting your entire Clojure program.

[Time 1:10:27]

What tools did you use for branch coverage?

It is a manual process with my editor and the REPL, with no fancy tools involved.

Workflow with spec and the REPL.

[Time 1:13:38]

When you save all code you have typed, is it anything like literate programming, e.g. using Emacs org mode?

I find it very difficult to work in a literate programming style. I tried org mode for Clojure code for a while, but now just use a plain text file for all of it. Promoting is by copying and pasting text, and then maybe changing names of things and wrapping a namespace around it.

What about state like connections, web APIs, data formats that change over weeks, months, etc? Discussion of examples of APIs that change way too often, and others that are much better.

Related advice: Strive to avoid creating dependencies on other code. Lucene library is huge, but does this well.

[Time 1:23:56]

Examples of multi-REPL use?

The one place where I tend to use multiple REPLs is with distributed systems, so one REPL per node I am interacting with.

Sometimes even a single REPL and just switch what it is connected to dynamically in the same window.

[Time 1:26:15]

Is there something like a tmux for REPLs?

I am lazy. I could use to learn many advanced tools, but prefer to use the simplest tools to solve the problem.

[Time 1:27:34]

Python vs. Clojure, what about these huge C libraries such as TensorFlow, etc.?

I think we should fight battles we can win, which is where there are already high performance Java libraries. If there are no high performance Java libraries, then we should not try to compete against Python in that, unless you want to be the person to go and write the high performance Java library, too.

end

[Time 1:29:40]