CaSE podcast - Problem Solving and Clojure 1.9

- **Speakers: Joy Clark interviews Rich Hickey**
- **Meeting: Conversations about Software Engineering (CaSE) podcast - May 3, 2018**
- **Audio and show notes: https://www.case-podcast.org/20-problem-solving-and-clojure-19-with-rich-hickey**

Original transcript: https://www.case-podcast.org/20-problem-solving-and-clojure-19-with-rich-hickey/transcript

[Time 0:00:00]

Joy Clark: Hello, everyone, and welcome to a new conversation about software engineering. This is Joy Clark, and today on the CaSE Podcast I'm going to be talking to Rich Hickey about Clojure, and all of the reasons why Clojure is the way it is. Thank you, Rich, for joining me today.

Rich Hickey: Thanks for having me.

Joy Clark: We've had two episodes on Clojure so far - one with Alex Miller about Clojure, and one with David Nolen about ClojureScript (we'll link those in the show notes). In this episode, for all the listeners who aren't that familiar with Clojure, they can go and look at those. But in this episode I want to concentrate on the Why question. You are the creator of Clojure, so why did you write Clojure?

Rich Hickey: I wrote Clojure because if I had to continue programming in Java, I wouldn't still be a programmer; Java, or C++, or C#, or the languages I was using in consulting at the time.

I had become an independent consultant and I was still working primarily in C++, but I got to do some work in Common Lisp and realized how I'd been wasting my time for over a decade, and sort of got the bug to figure out a way to use a language like that professionally. And through many paths that eventually led to taking a sabbatical and writing Clojure.

Joy Clark: What problems was it designed to solve? Why is it designed the way it is?

[Time 0:01:45]

Rich Hickey: It's fundamentally designed to reduce the complexity in programming. I think programming is far too complex, and there are a bunch of things that make it so. The domains are complex, and there's no way to get around some of that, but we have a lot of self-inflicted complexity. In particular in languages that aren't functional, the number one source of complexity is the use of state, so I knew I wanted to write a language that de-emphasized programming with "places", as I like to say (memory locations) and emphasized programming with values, like other functional languages.

And then there are other aspects of complexity that come into programming with any of these more commonly used statically typed languages, which have to do with the complexity in the language space. I think there's a lot of just very idiosyncratic stuff in programming languages that is unnecessary, and not only is it unnecessary and causing complexity in terms of taking up your mental space to understand it and apply it, but it also has other costs in terms of yielding programs that are larger, full of more concretions, less general, and more brittle.

Joy Clark: We've had some episodes about Clojure running on the JVM and on Javascript. What other platforms does it run on?

Rich Hickey: There is a CLR port. I believe that's the only near-complete version of Clojure.

[Time 0:03:36]

Joy Clark: Do you think there are any other platforms that it would be cool for it to run on in the future?

Rich Hickey: I don't think so. Obviously, linkage to C is interesting, but I don't think there is as well-defined a library space there as there is for Java and .NET, and that was the main reason to target those.

Joy Clark: I heard there was some kind of Erlang port. Is that a possibility, that that would be supported in the future?

Rich Hickey: Not by me. I'm happy to see people try things, but I'm completely uninterested in that. I don't know of any phenomenal Erlang libraries I'm dying to use.

Joy Clark: Do you regret putting it on the JVM? That's the main Clojure – because people who use it are probably Java programmers, or are coming for that direction.

Rich Hickey: That's not actually true. We get as many people coming from Ruby and Python as we do from Java. We get plenty of people from Java. But no, I don't regret it at all. It's probably one of the most important things to it having succeeded, because when people start with a new language that doesn't have libraries, they wait a long time to get up to the power level they expect by having access to libraries. And Clojure had libraries the first day. The very first day you had Clojure, you could use any database in existence, because JDBC existed and Clojure could use it.

[Time 0:05:13]

It's quite critical to the design of Clojure that it be hosted, and that it be hosted on a platform like Java. So I don't think that its interaction or dependence on Java (or Javascript, for the ClojureScript case) is a negative. I think it's a big positive. It does add something to what you'll have to become familiar with in order to be most productive. On the flip side, you can be most productive. That's the point. As a pragmatic point, having access to libraries means you have power.

Joy Clark: Clojure 1.9 was just released. What's new in Clojure 1.9?

Rich Hickey: At the language level it's mostly bug fixes and things like that, but one of the things that we've been working on is something called Spec. Clojure.spec is a library for expressing what are programs supposed to do, and do they do it? Like many things that come into Clojure more recently, it is a library. Clojure is a Lisp. As a language, by design, it's supposed to stay small. We shouldn't be modifying the language to accomplish new things. But we had some architectural things to do so that Clojure itself could use Spec in its implementation a little bit, and that led to a bunch of work that is maybe hard to see, but quite important, around how Clojure consumes dependencies and things like that. So that's probably the biggest thing in 1.9, in the language.

[Time 0:07:05]

The other thing that came out with 1.9 is a set of tools. This is the first version of Clojure that actually has a brew installer, so you can brew install Clojure. And the other thing it has is a runner, which can analyze dependencies, and create a classpath, and actually procure those dependencies – create a classpath and run your program.

And this is something that people had gotten from third-party tools, but we never really had a solution for newcomers who wanted to grab Clojure and use libraries on the first day.

Joy Clark: So the intention of the installer script is to make it easy to install on your machine, correct?

Rich Hickey: So there's two aspects. There's the installer, which is just a matter of obtaining Clojure. The more important thing is what you get when you run that installer. And what you get are a set of command line tools that do this dependency-aware invocation of Clojure. So if you want to run anything on the JVM, you have to eventually call Java and provide it with a classpath. And Clojure works by just supplying only Clojure on the classpath. But when you want to use libraries, they have to be put on the classpath, and you have to get them. And the whole getting libraries is a task that people usually rely on something like Maven to do for them.

[Time 0:08:29]

And then classpath building is something that, again, people had to go to other tools outside of Clojure, like Leiningen or Boot to do that. So the idea of getting started was “Get Clojure and get something like Leiningen and Boot and learn all of that.” So now you don’t have to, because the Clojure command line tools themselves can find libraries, download them, and incorporate them in the classpath.

Joy Clark: So Leiningen and Boot are like the Maven equivalent in Clojure, basically. So does the new installer replace that, or is it intended to replace it?

Rich Hickey: No, not exactly. I think one of the problems is: what “it” is. Those things (and Maven) do a lot. Most of them allow you to express your dependencies, and they can procure dependencies, and they can form the classpath, and they can be a runner and run Clojure with that classpath, but they also can build generally, and/or deploy, and make uberjars, and all of that. And that aspect of Maven, and similarly in these other tools, turns them into kind of large, monolithic things. And another big part of what we’re trying to do with this dependency stuff is to separate out and decompose that problem and say “This part that we’ve delivered is strictly about dependencies and classpath generation, and it doesn’t do any building, or anything like that.” So you still might want a tool like that to build your final deployable thing.

[Time 0:10:16]

Joy Clark: Are there any ideas in the future about providing something similar for ClojureScript, so you can never have to go into npm hell ever again?

Rich Hickey: I think that the problem of dependencies is a big problem. It’s one of the unsolved problems right now for programming, and I’ve been thinking a lot about it. And this is on the path towards that. One of the other things that you can do with this new dependency stuff is directly consume code from Git repos by using Git SHA’s as your dependency, as opposed to an artifact in a place. It’s in addition to that. It can consume Maven still.

But the other thing that we did was sort of decompose or decomplect transitive dependency analysis, which most of these tools rely on Maven to do. But Maven only understands Maven and only navigates through Maven dependencies.

And we now have lifted the transitive dependency analyzer out so that it can traverse through Maven dependencies, but also through Git, and through more than one way of representing your dependencies, whether it’s a POM or a project config file, or the new format that these tools use, which is called deps.edn. So this is super important, and it takes us towards where I’d like to see us go, which is a stronger connection between what we’re consuming, and the source truth of that.

[Time 0:11:57]

I think a big problem is you grab a library and you really have no idea what you’ve gotten. Maybe it has a label. It says it’s 1.2, but you don’t know which functions inside it have changed, or why. You don’t actually know what you’re running. And maybe the jar file or artifact tells you something about the source that was used to produce it, but the process that was used to produce it is often opaque. And any of that could be wrong because there’s a lot of human steps involved in producing artifacts. And because so many people use Git, I’d like to get closer to leveraging some of the features there, in particular using SHA’s and content-based addressing to talk about things.

I had worked on a library called Codeq, which we’ll have a new version of soon, that sort of extends the Git model down to the function level, so you would have SHA’s for individual functions, and you could have dependencies on functions, instead of on artifacts. So a lot of work is happening around that, and some of that manifests itself in this dependency tool.

[Time 0:13:15]

Joy Clark: I'm thinking about dependencies and what happens when I upgrade my dependencies, like in my Java project, and I get a whole bunch of compiler errors. But in Clojure they would be run-time errors, right? How do you deal with that? Because Clojure is a dynamic language – it's not compiled – when you upgrade a library and it's incompatible, how does Clojure know what happens then?

Rich Hickey: Well, there's a lot of presumptions in that question.

Joy Clark: There are. There are a lot of presumptions.

Rich Hickey: The first thing I would say is what you're talking about are symptoms, right?

Joy Clark: Right.

Rich Hickey: The problem is not that the problems are found later or earlier, or that they manifest themselves as compilation errors or run-time errors, but the fact that your library provider broke your program. And that's something I don't think should happen nearly as often as it does, and is actually quite avoidable, but unfortunately we still do place-oriented programming in the library space. So you could say that library Foo is a place, and every time I look at that place, I find a different library. And sure enough, the values are just changing out from under me, and the things I depended upon are no longer true.

[Time 0:14:49]

When a library breaks, it can break in many ways. Some of those may or may not be manifest in types, others would just be manifest in behavior, or missing information, or additional requirements – things that you can't express in types, because most of what your program needs to do can't be expressed in the type systems we have today. So yes, it still takes a string and still returns a map of information, but it stopped returning you some of that information, or it started returning other stuff, or it had additional requirements about the string. The types don't capture that.

Joy Clark: Does Spec help with that problem, of knowing what actually goes into your function and what comes out?

Rich Hickey: So there are a couple of different problems here. If we stick at the library level, one of the problems is breaking things. And I think one of the problems we have is that we think about change generically, as if change was a thing. But there are really two very distinct kinds of change. There are breaking changes, where your expectations have been violated, and there are accretions, where there's just some more stuff where there wasn't stuff before. And in general, accretion is not breaking.

[Time 0:16:13]

So if I had a library I was using and it had functions A, B, C, and version 1.2 of the library added functions X, Y, Z, my code that used A, B, C is unaffected by that. So if you take that idea of what accretion means, you can now apply that to things like argument lists or return values, and you can start talking about what a function either requires, or provides.

So let's say I had a function – one of those A, B, C – that already exists and you're already calling, and I wanted to enhance it. Well, if I require more from you than I used to, I've broken you. If I require less from you than I used to, I haven't. On the return value side, if I provide more to you than I used to provide, I haven't broken you. But if I provide less, then I have – if I provide less than what I used to promise.

And so there's a real directionality to the contracts of functions and code, and there are changes you can make in both directions that are compatible with existing consumers, and changes you can make that are not. And when you understand that you're either adding requirements breaks and removing requirements doesn't, and providing more doesn't break, and providing less does break, then you can say: all the non-breaking changes are evolution-compatible. They allow programs to evolve. They allow them to evolve independently, which are super critical properties for systems to run over a long period of time.

[Time 0:18:07]

I think that it's a big mistake to say "Well, static types allow me to break people, and they can figure it out." That's exactly what you don't want to do. So I would say that if you're going to break someone at all, just don't, and call the function D, or A2, and leave A around. These are the same kinds of strategies that we use at the service level in order to have loosely coupled systems that don't break consumers, and allow clients and servers to independently evolve, and it's just as important in the small.

I think that it's, again, sort of a big problem that we think about change generically, and we think about typing tools as tools to enable breakage. And that makes for brittle programs. And it was certainly my experience before Clojure, working with static type systems, that all the systems were incredibly brittle, and eventually the cost of change just got so high. Every system eventually got thrown away because the cost of change became too great.

[Time 0:19:25]

Joy Clark: So in these functions, you would take a map and return a map? Because I can imagine that even if I were to change a function and say "Oh, I need something else", if I change the arity of the function, that's also breakage. I mean, I can put another function next to it, or make it a vararg function. But essentially, every time I need something new, I personally would go for creating functions with a couple of arguments, as opposed to putting everything in a map, but maybe that's not the best way to do it.

Rich Hickey: So it ends up being the case – and this directly connects to Spec and to your question from before. You know, is Spec in this space, trying to contend with these challenges of allowing you to talk about what your program does and determine that it works, and allow you to communicate to consumers about your contracts?

It does, in fact, and the way it does is: it allows you to talk about either function signatures or data structures using one of two logics. One would be set logic around maps and map keys, and the other would be regular expressions for things that are sequential, like function argument lists are an example of a sequential contract.

[Time 0:20:52]

And it ends up in both spaces, those ideas I talked about before – about requiring less or providing more – both apply. If you think about a set of keys, if you're accepting that as an argument, then to require more keys would be breaking, but to require fewer keys would not. If you're returning maps, then to return more would be non-breaking, and to return less, at least about the things you promised you would return, would be breaking. So the same ideas apply. So you can talk about maps and you can – well, we're still working on the language to make it precise to talk about providing and requiring certain keys.

The other thing is in the regular expression space there's also a notion of regular expression compatibility, which is to say there is logic behind regular expressions that allows you to say that this regular expression can accept all things this other one could, but maybe some additional things. And you could leverage that logic to actually make non-breaking changes to function arity. For instance, if you could take two arguments before, and now you could take two or three, you haven't broken anyone, as long as you haven't changed the meaning of what it means to pass two. You've just enabled people to be able to pass three.

[Time 0:22:20]

And I think it's essential for people to start understanding what it means to provide or require, and to look at their changes in that light. It's certainly one of the long-term objectives that for Spec we will take the expressions you make in Spec about providing and requiring, and turn that into a test for change, to say "If I want to modify this spec and it's a spec about requiring, is it compatible or is it breaking? And if it's breaking, we won't allow it." And same thing flipped around on the return side.

Joy Clark: What does Spec look like?

Rich Hickey: Spec is a predicative language about data. So much in Clojure is data. We express all of our information as generic data. We write code as data structures. We use data structures as domain-specific languages for configuration, or HTML, and pretty much everything.

So the idea is to have an open system that is not limited to any particular logic or static verifiability, but to say: any predicates you want, you can write regular Clojure code, and you can basically say what you consider to be required to be true of a data structure. And you do that either in the small, with assertions about the types of atomic things, like numbers, or booleans. And then in the large, you use one of these other two techniques, talking about either maps and the sets of keys associated with maps, or with regular expressions for sequences.

[Time 0:24:23]

And most of the sequences – people tend not to use sequential things in wire protocols and service contracts, because they're terrible. They're very brittle. But we do still use sequentiality in function signatures, so it's necessary for that.

So you write those expressions, and it's just like a little domain-specific language. But it's independent of the Clojure code. You can write specs for things that you wrote, or things that other people wrote and they haven't spec-ed themselves. If you're trying to understand a library, you can write some spec expressions about it and see if they hold. And the first job of Spec, given one of these specs, is to do validation, which is what you would expect. "I have this predicate about this data structure. Is it true of this particular value?"

But Spec does a lot more, because it also does generation. So it can generate data satisfies the predicate. So you can say "I have this function, and I said it returns this shape", and you can say "exercise this spec" and it will generate a bunch of those shapes. And that's the underpinnings of the next feature, which is that Spec supports generative testing, QuickCheck style testing. So if you spec some or all of your code – and it's not a type system. It's not necessary for it to be complete or for there to be types everywhere. It's not like types. It's really predicates. But for whatever predicates you've defined on functions, Spec can automatically generate data to test those functions, taking it through a random space, and validate the function's work, that they are returning what they say they will.

[Time 0:26:28]

And remember, these are arbitrary predicates, so the predicates of what you return can include what you put in as part of the argument. So you actually can spec the behavior of a function, as opposed to just: its return value has some particular shape. You can for instance make sure that, if you were passed a collection coming in, every member of that collection coming in is present in the collection going out. And various other value things like that.

So the idea behind the testing is that these validations and these tests are not for production run-time. All the tools are set up to allow you to do this work prior to release. That it's part of the development time, and testing phase of development. And all the checks and everything are turned off at run-time. You can still use Spec if you want to, to be a gatekeeper on the end of a wire, for instance, because you can use Spec to define specs for wire protocols and communications protocols. So that's something you might want to leave in in production. But otherwise, there's no overhead associated with using Spec.

Joy Clark: So at production run-time you assume that you've done so much generative testing that the functions must be the correct specifications, so we don't have to do the same checks during run-time, at production?

[Time 0:28:08]

Rich Hickey: Well, you can make whatever decisions you make ordinarily, but it's no different from static systems. They have tests too, and they can either run those tests all the time, or just before they ship. But what you can express with Spec is way more than you can express with a type system.

Joy Clark: Is it sufficient – because I came from a background of doing a little bit of like formal validation and specifications for safety-critical systems. Would Spec be helpful in that context? Because that's one area that I personally find a type system, or some kind of proof – I want to have a proof that my program actually does what it says it's going to do, because someone's life is on the line. But would Spec be a valid –

Rich Hickey: I consider that pretty much hyperbole. So let's take the function "reverse". So in Haskell, reverse has a type of "list of A to list of A". If that type checks, do you know that that works?

Joy Clark: No.

Rich Hickey: No. And that's true of most of your statically typed software. Most of what's important about what they do is not captured by the type systems, because they are not semantic in the first place. They're mostly just mechanical, and the logics they have are pretty weak.

[Time 0:29:38]

I think if I had a safety-critical system, I'd be looking outside of a type system – because they're so anemic – to a stronger formal verification system that is outside of the program. If you wanted to write an algorithm to make sure you were doing spin locks correctly, or distributed transactions correctly – people do use proof systems for that, but they don't use the ones built into programming languages. They use much more powerful ones.

Joy Clark: That is true.

Rich Hickey: I'm in favor of that. I'm mostly advocating that, like Spec, it be à la carte.

Joy Clark: Yes, okay. You also created Datomic.

Rich Hickey: Yes.

Joy Clark: Could you talk a little bit about just what that is, and what its current state is?

Rich Hickey: Sure. So Datomic is somewhat pointed at the same problems that Clojure is. Things are too complicated, and we're doing place-oriented programming. And the other big place you have left, once you switch to a functional programming language, is your database. So you can do whatever you want, you can use Clojure or Scala or Haskell, and then this database ruins everything for you, because it is a place, and most databases update in place. So there's all the complexity associated with that, that there is with using places in memory.

What Datomic endeavors to do is to say "Let's stop doing that. We have a lot more storage space than we ever did. And we have enough that we could take a functional approach to storing our data."

[Time 0:31:27]

I think people already realize this. People are certainly logging everything and keeping everything and going append-only. The real value proposition of Datomic is to both, again, sort of only accrete information, not change it in place, but also provide the logical support for accessing that. And in particular, for allowing you to use the database at a point in time as if it were a value in your program. That is the big trick of Datomic.

When you do that, a lot of things that were complex become straightforward, just like when you move away from mutable variables to values. You're able to say "Of course I can perform three distinct operations with this value, and not worry about the consistency of their results because the value might have changed." That is a hard thing with update in place databases. You do three queries, and if stuff happened in between, then the basis for the queries was different, and their results can't be correlated.

With Datomic, even if things have accreted in between your three operations, you could treat the database as a value and perform three operations, and know you get results that are consistent between them.

So that's the idea behind Datomic, and the state of it is we're right on the verge of shipping the cloud version of Datomic. So it had been so far that you run on your own behalf, with a traditional install, and now we will have an offering in Amazon's AWS Marketplace for getting Datomic that way. You'll have the ability to get a Datomic instance up for around a dollar a day to get started and explore it, and then we have different production levels of instances and deployments.

[Time 0:33:30]

So we're hopeful that will make it a lot more accessible, and certainly help people program for the cloud. I think, again, we see in that space a lot of complexity, as people take software that was written for a stable network, with machines you could go and put stickers on with pet names, and kick, and unplug, and plug back in, and replace the hard drives in, to this more ephemeral world of the cloud. And as you move data center software up to the cloud, you struggle with a lot of things.

Datomic Cloud was written for that environment, so it's completely tuned for working with Amazon's logging, and metrics systems, and encryption systems, and things like that. So we're hopeful that will make Clojure a really good language for doing cloud development, because you'll have this tool that's compatible with that.

Joy Clark: Awesome. I'll have to check that out. In your opinion, is there a benefit to opinionated approach to program architecture? I have some people who talked to me about Clojure and they're coming from the Rails world, or the Spring Boot in Java, and they're like "How do I write a web application in Clojure?" and I'm like, "Well, you can use all these different libraries, and they compose together, but you can pick and choose." And I think one of the reasons it's difficult – well, it's not that difficult, but it's a reason that makes it more difficult to get started, because there's just so many options. So is there a benefit to having a standard stack, where you can say "This is what you should use?"

[Time 0:35:27]

Rich Hickey: Well, some parts of that question are social, which I can't really speak to. I think certainly when somebody figures out how to do web development, they should encode it in a framework. But I'm not sure that that's a solved problem. And I think until it is a solved problem, opinions are very much opinions, and therefore you're at risk adopting a set of opinions that may not be an answer. I certainly let the community find its own way in this area. I think we have some very talented people doing really good work, including David Nolen.

I think that it's really important to be able to say "I don't know yet." I'm not gonna go code up this big thing, because you know, some of the things that you're talking about as being standard, they also have a bunch of known shortcomings. So while socially it's straightforward to say "Everybody's doing X", you may not like doing X after you've done it for a while and found all their problems.

Joy Clark: That kind of leads into my next question. What other problems do you think there are left to solve?

[Time 0:36:53]

Rich Hickey: Well, I certainly think this dependency problem, and program evolution, is a big question. We certainly build programs out of parts, and now we build systems out of programs. And so the whole idea of programming – you know, in the old days a program was completely self-sufficient, almost used no libraries, it was written by one person, ran on one processor, and knew everything. Now a program is just a dot in a big picture of the system. And there are many programs, maybe more than one programming language. There's a database, or more that one database. There are wires, and wire protocols and things like that.

And I think that when you think about that kind of a system, you end up with a bunch of pressures and concerns around things like what I was talking about before. What are your promises? What do you require? What do you provide? How do you talk about them in a way that supports evolution?

Because it's simply insufficient to say "Today we know the type of everything, and we've encoded it all, and we've all agreed, and we've all recompiled, and it all works perfectly." And then tomorrow, if anything changes, everybody needs to stop and rebuild. That doesn't work. That's not the way the internet was built, and it's just not the way bigger systems work. And I think we're suffering in the small, from not paying attention to those same concerns. Like you said, feeling like it's a regular thing to have a library break you, and thank goodness your type system helps you figure out some portion of how you got broken, maybe. That shouldn't be happening.

[Time 0:38:52]

I think that's a big area. To the extent that we had relied on our languages as sort of tools for encoding our understanding of the problem we're trying to solve, that's obviously inadequate now. Because the programs are always sub-problems, and it's more important to talk ... It's way more important for an architect to be talking about the schema of their database and their wire protocols than it is about types in any programming language. It doesn't matter which.

So when do you start talking about that stuff? How do you talk about it? What's nice about something like Spec is: it's just as good at talking about wire protocols and Datomic schemas as it is about Clojure. In fact, it really has nothing to do with Clojure. And it's things like that I think: that's an area for growth. We need more of that.

Joy Clark: Do you think it's actually going to be solved? I'm just thinking. Maybe I'm just too much of a pessimist. Maybe I just always expect that someone will go and break the library at some point in the future, and it's all futile.

[Time 0:40:15]

Rich Hickey: Well, I think it's hard, because until you're able to disentangle breaking change from compatible change, you sort of can only shrug at this problem. But if we had more tools that could say "No, no, no. This change is a breaking change, and this change is a compatible change", then it wouldn't just be this black box of change. Similarly, if you had more insight into the granularity of changes in a library, as opposed to just knowing this giant artifact, with all its dependencies, moved from 1.2 to 1.3, and just – who knows what's different? ... to actually know.

And again, things like Spec, and specs, and things like Codeq, that give you more insight to dependencies at finer granularity will at least allow you to start having conversations with people about what they're doing, and what you're doing, and what the promises actually are. But a big part of it is certainly social.

I think one of the things that's been nice about Clojure - because you've talked about opinionated, and I think you meant more in the area of framework. But I think certainly Clojure is opinionated in its design, and what it makes idiomatic, and what it makes hard, even. And I think that that does help people make decisions. At least they have 1) an example, and 2) hopefully, reusable tools they can use to emulate the example. When you have more of that, I think you make it easier for people to do the right thing. And I guess that would be another objective of Clojure, would be: to help make it easier for people to do the right thing.

[Time 0:42:16]

Joy Clark: But that can't be solved by one programming language, because there probably will not be a homogeneous programming environment, like one system is written only in this language. I mean, I don't think Java is going anywhere anytime soon. There will probably be people writing Java applications that we have to take into account in some way or shape.

Rich Hickey: Right, so where programs meet each other are on wire protocols. We're not using Java serialization over wires, so what are we saying there? What are we saying about that stuff?

People are trying things. I think a lot of what they're trying has a lot of the shortcomings of static typing in terms of not being evolution-oriented, but I do think you'll see more work on that. Certainly, other things that we've done in the Clojure dev space have been around that, defining something like edn, and things like Transit, which are wire protocols. It's part of the job. It's not about Clojure the language. It's about tools for communicating. Spec will be something that can work on wire protocols. There may be other things.

Joy Clark: Is edn used in other programming languages besides Clojure?

[Time 0:43:51]

Rich Hickey: Edn is not, for reasonable reasons. I think that if you're going to solve the protocol problem, you have got to be able to reach the browser, and that's why we worked on Transit. And Transit is actually quite good at communicating between languages and to the browser, so the performance of Transit is at the browser–

Joy Clark: What is Transit?

Rich Hickey: Transit is just another way to encode the same set of data structures. So Edn can encode the data structures of Clojure, which I consider to sort of just be the basic data structures. You have some atomic types like integers and floating point numbers and booleans and strings, and then you have maps and sets and vectors and lists. So edn can encode all those things. Transit can encode those same things, and also, like edn, is extensible in that you could define new things in terms of the things that are built in. So if you wanted to have a record type or something like that, you could define that.

What Transit has over edn is: edn is sort of a character format, and Transit has encodings, in particular, directly to JSON. But it can be faster to transfer stuff through Transit over JSON than it can to transfer JSON sometimes, to the browser, because Transit incorporates things like redundancy compression and stuff like that.

So that's an effective thing, and I know people are using it from different languages. That's not to say it's as popular as protocol buffers or something like that, but I do think that this is the area, this is the space - wire protocols, and also protocols or specifications for services. How do you talk about what your service expects, what it requires, what it provides, in a way that is compatible with evolution?

[Time 0:46:00]

Joy Clark: Yes, I'd like to have that. I love the "Oh, we changed the format of the date string", so now my program broke. Thank you.

Rich Hickey: Yes. But we're having a conversation right now about what is a breaking change or not. Ten years ago, the conversations we were having about functional versus mutation seemed as novel, and now it's quite common. I think that even in languages that are not primarily functional, the emphasis on a functional approach is widespread. We've certainly seen a lot of the Clojure ideas incorporated in the Javascript community, and that's not just because David Nolen is so charismatic, but because they work.

So I'm more hopeful. I can't solve the social problems, but I think we can make tools and make examples that make doing the right thing more straightforward.

Joy Clark: I think it's interesting, because I started programming maybe 7-8 years ago, and I was kind of brainwashed into believing that immutability was the only correct way, like from very, very early on. And so I don't understand object orientation really, and I always try to – like, I have to program in Java, and people are sending me things and like "Why do you do it like that? I don't understand. You could just make it public, and final. It doesn't have to change." So I just don't know. Yes, I don't understand object orientation. That was more of a comment, that wasn't a question.

[Time 0:47:55]

Rich Hickey: Yes. Well, that's the thing. I think as long as people are writing functional languages and writing in functional languages, they will serve as examples, and some younger programmers will never have seen something else, and won't have to unlearn a bunch of bad habits.

Joy Clark: Yes. I think it's interesting, because I don't know anything else. Sometimes people ask me why I like functional programming so much, and I'm like "Well, you know, it just feels right." I don't have a good reason, I can't argue it from the perspective of someone who came from object orientation, and apparently there's a whole bunch of problems with that. I just usually say, "Oh, I like it." I don't really know. I can't argue the point as well as some, maybe, for that reason.

Rich Hickey: It's worth learning enough about the other to argue the point objectively, so it isn't just a preference thing.

Joy Clark: Yes, that's what I think. I would like to do that. I don't want to just choose something because everyone is doing it. I feel like functional programming is a trend now, and everyone is jumping on the functional programming bandwagon. But I would like to have a reason behind that, not just "Because everyone else is doing it."

Rich Hickey: Functional programs are simpler, for the deep definition of simple.

[Time 0:49:44]

Joy Clark: Yes. So you've challenged some of the conventional ideas of programming, like the reliance on mutable data, and changing data in place, but also practices like agile software development, and the obsession with testing. What's your view about the ideal way an efficient team of developers should go about programming?

Rich Hickey: They should spend a lot of time thinking about what problem they're trying to solve before they do anything. That is the number one problem in programming - the time is not spent, and people flounder around. If you did that, you could probably do well with almost any set of tools. I don't make tools that are sort preeminent in programming success. I think if you really are good problem solvers and are focused on the problem, and decomposing the problem(s), you're going to do well.

I don't think that people should write a lot of tests. I think people should run a lot of tests. We don't have better technology than testing for determining that our software does what it's supposed to do, from a real-world requirements perspective, what it's supposed to do. So we need to be able to say what it's supposed to do, and then hopefully get some technology helping us to determine that. And the best technology right now for that are things like QuickCheck, that kind of generative testing.

[Time 0:51:34]

So instead of writing a test that is an example of one successful execution, what you should write are properties – or specs in the case of Spec – and let the computer write a million tests for you and run them. So I'm a big proponent of testing. I'm not a big proponent of test writing. And the way people test I think is pretty terrible, until you adopt generative testing and property-based testing and specification-driven testing, where there's a tremendous amount of power.

So I think you want to adopt that. That would be my next piece of advice. Certainly, you want to be doing functional programming primarily. Not that you can avoid doing anything else at all times, but that should be the default, for sure. And I think that you should be working in a language that makes those things idiomatic, so that you're not fighting against a different paradigm. So while people can do functional style Java or Javascript, it's never going to be idiomatic. It's always going to be hard. It's always going to look awkward, feel awkward, confuse your co-workers. So these languages have a lot to offer.

[Time 0:53:08]

Joy Clark: Are there any signs – if you're solving the wrong problem, how do you figure that out in enough time that you can change, and solve the right one instead? Because testing doesn't prove that I'm solving the right problem.

Rich Hickey: No, not at all. Yes.

Joy Clark: It proves that what I did, the code that I wrote, does what I think it should do. Which is good, but if I'm solving the wrong problem, it's all wrong.

Rich Hickey: Yes, so there are two things. It could be just the wrong problem in general, or you haven't decomposed the problem well. I don't know of any magic recipes. You're outside of technology at that point. It's funny, because I think programmers – especially young programmers – they use far too much of their brain power on programming stuff. Learning the idiosyncrasies of a programming language, learning some really arcane type system stuff, or tons of details about some gigantic Javascript UI library du jour, and not nearly as much time on the domains in which they operate.

[Time 0:54:37]

One of the coolest things about programming – and I tell this to people who want to take up computer science – is that you really get two specialties, or more. You get to learn about computer science, and programming, and things like that, but you also always are applying it in a domain. And unless you're a compiler writer or a theorem prover writer, you're going to have some other domain. You might be doing medical imaging stuff, or music scheduling, or writing an election system, or a climate analysis thing, or e-commerce. And these are big, fascinating domains with lots of interesting aspects.

I think a good programmer is highly engaged with the domain side of what they're doing. And so I'd rather see people spend a lot more time there. And that helps you with the "Am I solving the right problem?" How much of your brain time are you spending there? Did you spend 10 minutes with somebody in the domain, and then you think you're just smart enough to go run off and code it up? Or maybe you took everything they said at face value and never questioned it?

[Time 0:55:56]

A lot of times users in a domain say they need X, and a good programmer – or a good systems analyst was the old time name for it – would know how to turn that into a conversation where you would take a step back from saying "All right, I know you're presuming you need this answer, but let's roll back a little bit more towards your problem. And maybe if I understand your problem better, we'll have more than one possible answer", and those answers will have different tradeoffs or different characteristics.

That's where the work is. That's certainly where I spend most of my time. I would rather chew on a problem for ten days out of two weeks and code for the other four, than to agilely rewrite it six times in four weeks, and then still have it be wrong.

Joy Clark: Are there any technical things that are worth looking into? I feel like a lot of times the younger programmers come and they're like "Oh, I have lots of great ideas!" and they do exactly the same errors as the people who were learning to program 15 years ahead of time. And I know there are some great ideas from the '70s and '80s, but I never know if – I guess my question is: I have a lot of time ahead of me personally, and I know I have a lot of brain time ahead of me, which I'm looking forward to, but if you could give some tips about what direction I should use my brain energy on, that would be good.

[Time 0:57:43]

Rich Hickey: Well, people talk about reading papers and whatever, but I certainly would point anyone who wants to learn more about programming to the papers from the old days of computer science. From the days of Lisp and Smalltalk, when the people who were writing computer science papers were also writing systems. They were trying to write logic systems, or reasoning systems, or chess playing systems, or user interfaces, or the first databases.

The earlier computer science papers – not all of them, but a lot of them – from those communities especially, are very special in that you've got the work of people who are ... Obviously, they're still researchers and they're still doing theoretical stuff, but the level of practice that was present there is quite evidently different than at least what gets published today, which is just math. So there's a lot of systems stuff that is really fascinating, like reading about Self, and what they were doing to try to optimize dispatch. It's fantastic, really cool.

Joy Clark: Do you know them off the top of your head? Could you name some papers to look into?

Rich Hickey: You can find the Self paper.

Joy Clark: Okay, Self paper.

Rich Hickey: "Self" is the name of the programming language, and there's a good paper – probably in the history of programming languages. It's sort of a big summary of the whole project. But all those papers . . . The Smalltalk papers, and the Common Lisp papers, and the Lisp and Scheme papers, are all really good.

[Time 0:59:32]

Joy Clark: Well, I will try to look for them and put them in the show notes for any listeners. Thank you so much for your time. I had a great time talking to you.

Rich Hickey: Yes, thank you. It was a lot of fun.

Joy Clark: And thank you for creating Clojure.

Rich Hickey: You're welcome.

Joy Clark: Okay, so to all our listeners, until next time.

[Time 0:59:56]