This is a keynote from ElixirConf 2015.

Elixir is a programming language (and tooling) that compiles to Erlang

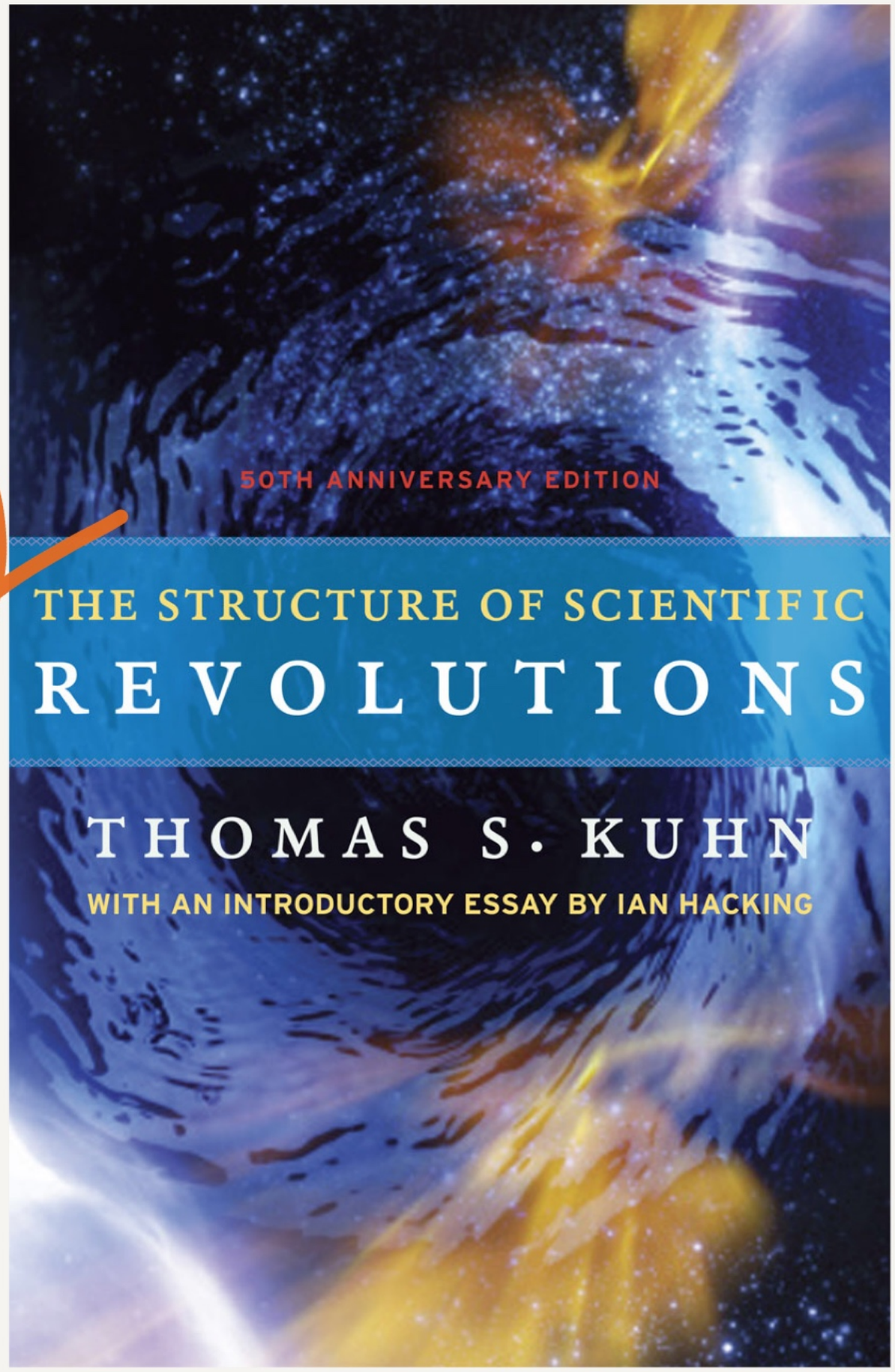
and runs on the Erlang VM. It is a language designed to be pleasant to code it. Elixir is functional, with immutable state. It runs on the actor model, with lightweight actors passing asynchronous messages to each other. This is native concurrency support, and native support for distributed systems.

It's kind of bananas to come to learn a language no one is clamoring to hire you for (yet). To come to a conference, taking vacation or eschewing pay. Travel on your own dime. All to learn and teach about a language like

Elixir. In the hope of being part of progress in our field.

It reminds me of science and scientists. I majored in Physics as an undergrad, then very rationally went into programming. Those who stay in the sciences through PhD and do research - they're not maximizing economic game. They're in it to learn, individually and as a society.

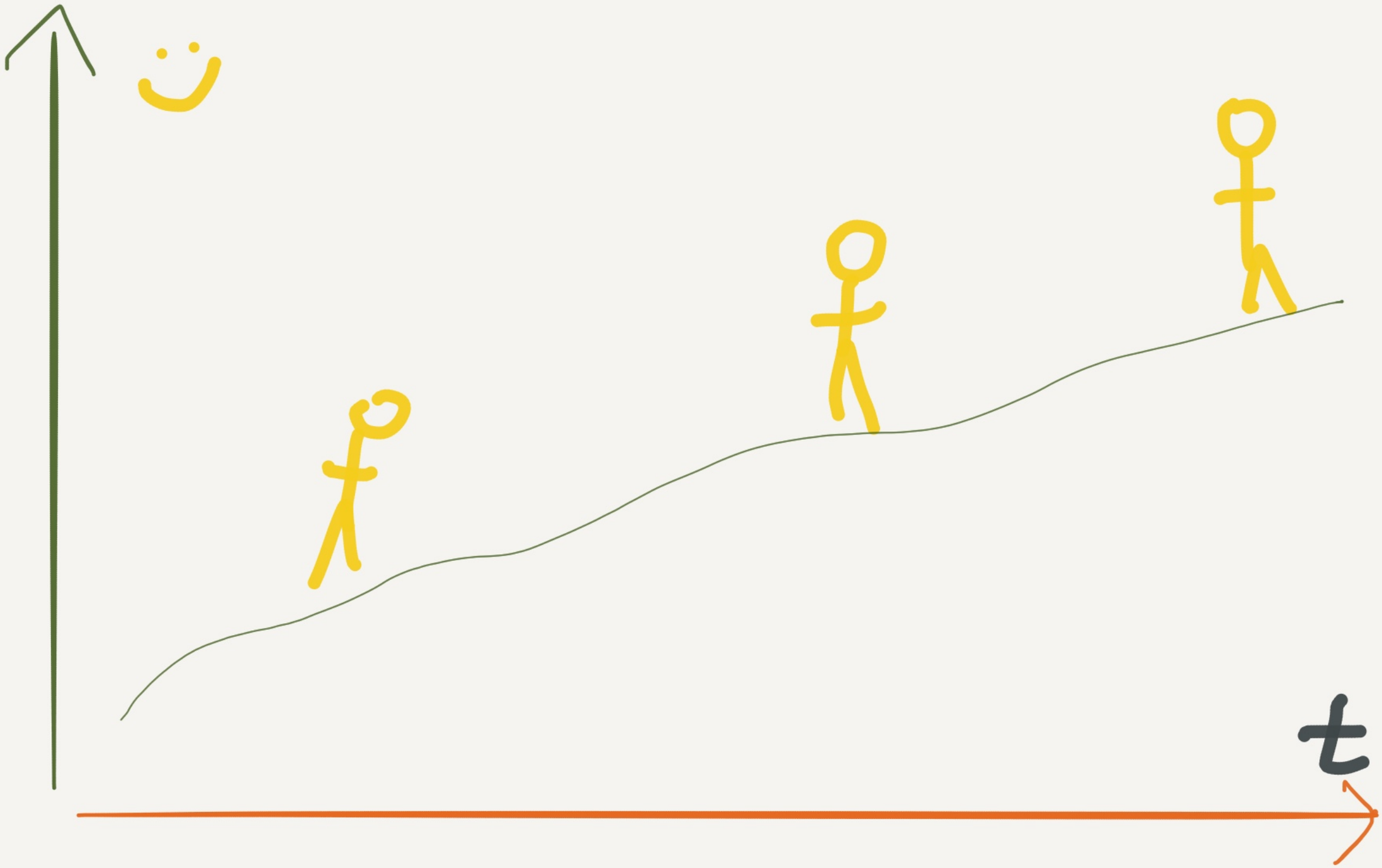
1962



This is a book about scientific progress and how it happens. It was written in 1962. I read it in 2012. It has a lot of insight into how science

moves forward. And how ideas move forward.

This book is the one that brought the word "paradigm" into common use. It describes the slow process of paradigm shift.



Scientific progress marched inexorably forward. One great man (they were always men in the histories) after another led us toward greater

knowledge.

That was the story until Kuhn dug deeper.

Kuhn points out that progress is in fits and starts. Many competing theories. Then one that works. Until it doesn't. Then there is a period of crisis until a better theory wins out.

For example, Electricity. In Ben Frankin's time, there were lots of theories. Most viewed electricity as an excitation of the matter in the object.

Franklin focused on conduction, like in the Leyden Jar (a device, the first capacitor, itself invented independently by a German and Dutch scientist separately at the same time). And he found lightning, identified it as an electrical phenomena. He categorized it with the little sparks that jump between your sweater and a balloon.

Meanwhile du Fay in France focused on electrical attraction and repulsion. He formed a 2-fluid theory.

1747, Franklin has the 1-fluid theory. A positive fluid, and we still think of it that way when we talk about conventional current.

Somebody eventually realized they were equivalent. Much later. This theory served for a good sixty years.

Anomaly: Ørsted noticed that a compass reacts to an electric current. This is explained neither by the fluid theory of electricity or the corresponding fluid theories of magnetism.

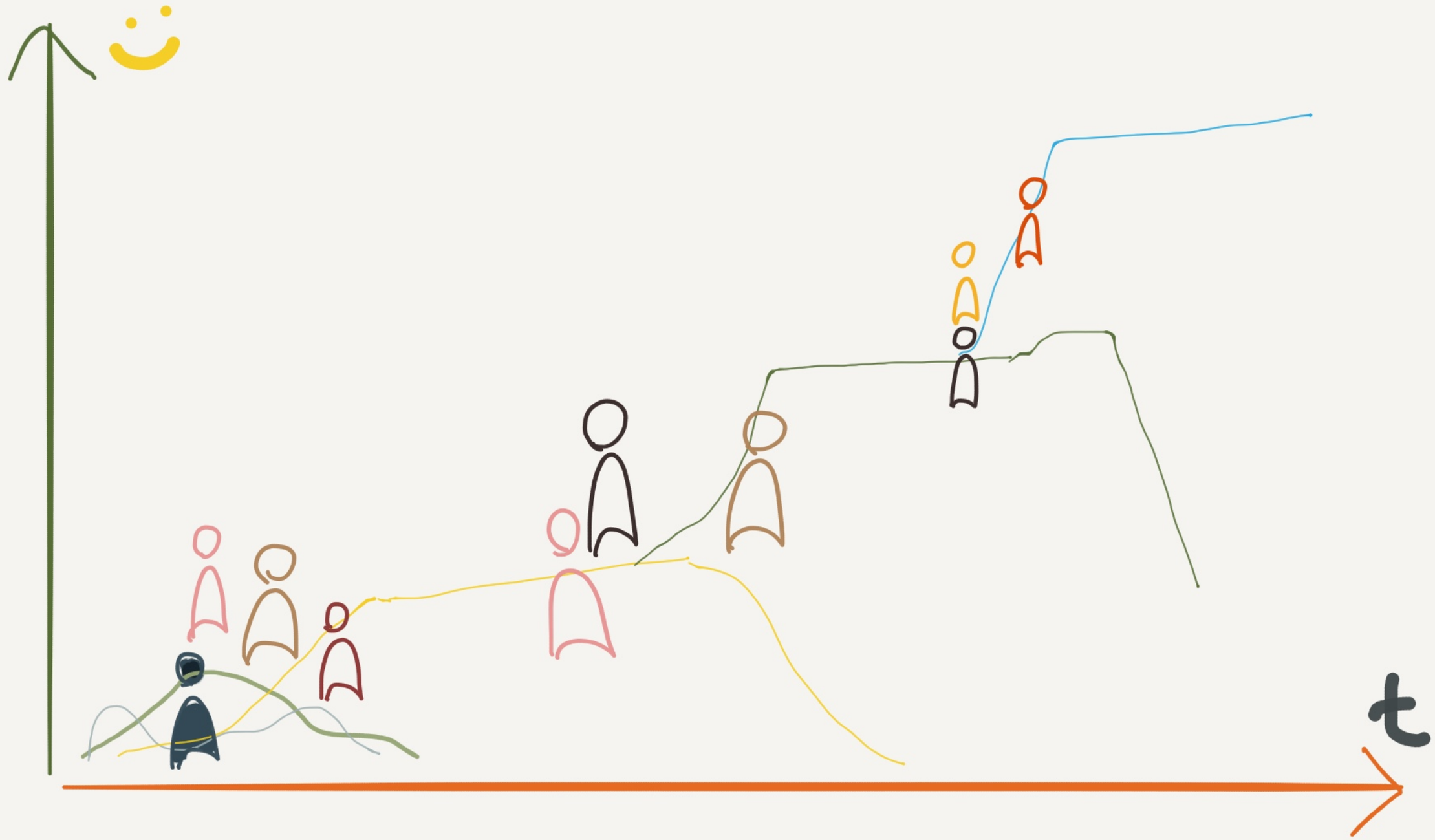
1827, Ampère unified the electrical and magnetic sides, and founded electrodynamics, by finding that wires with current attract or repel each other. He had the "electromagnetic molecule" part closer, too. But he still had it all mixed up with aether.

It is never this simple, though. Poisson, Biot, Pouillet, other scientists had competing theories. Ampère's collaborator Savary was essential to his work, did much of the tricky math while Ampère was busy. Ampère praised Savary but we don't remember his name.

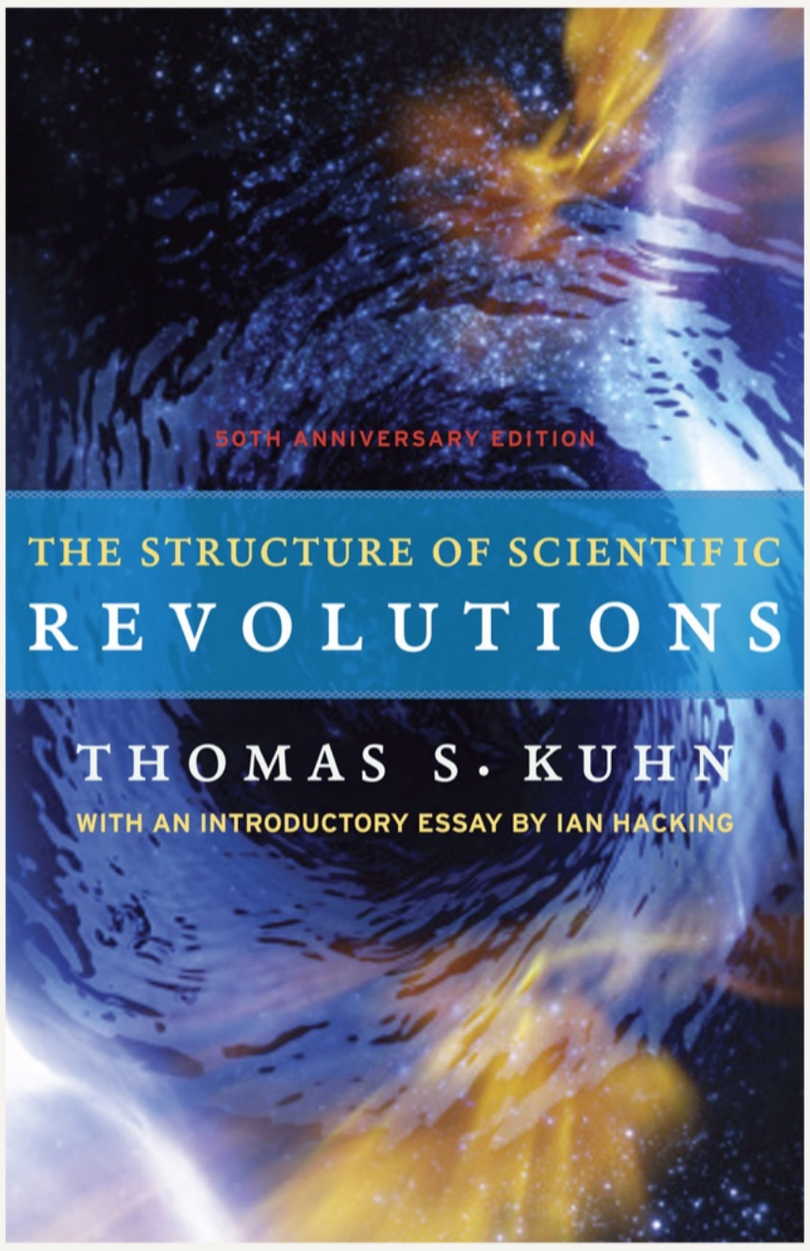
Ampere got in with the publishers and was able to publicize his work because of his relationship with Laplace.

During this period, French papers were full of "new electrodynamic effects" as everyone started looking for them. New theories abound.

Later Maxwell unified electromagnetism with light. That was probably messy too.



It isn't one person who discovers a theory. It's usually several. Newton and Leibniz. Gödel and Church and Gentzen.



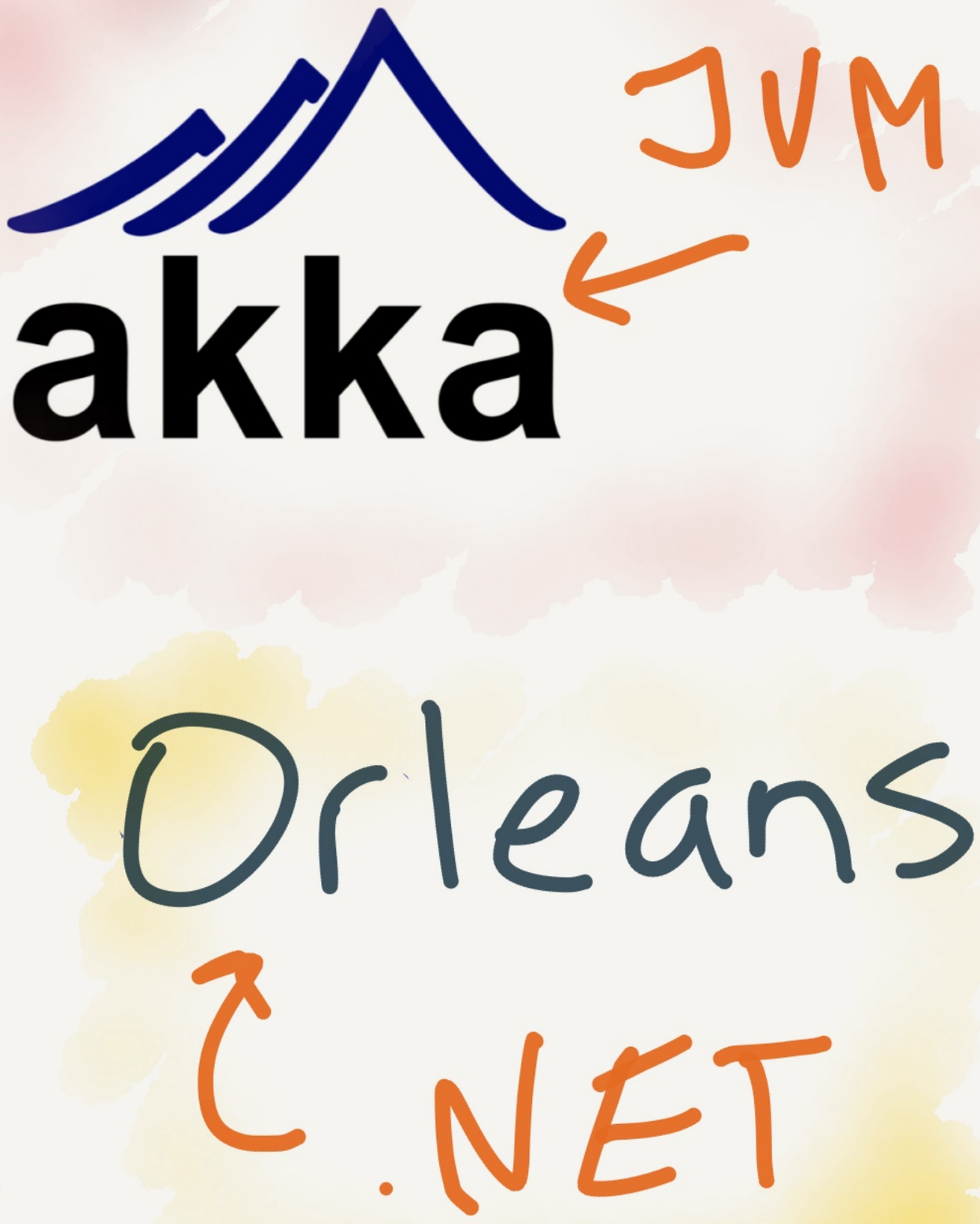
1. Ideas are shared

2. Ideas keep coming

3. Ideas are shared

Three things I learned from this book are directly relevant to software.

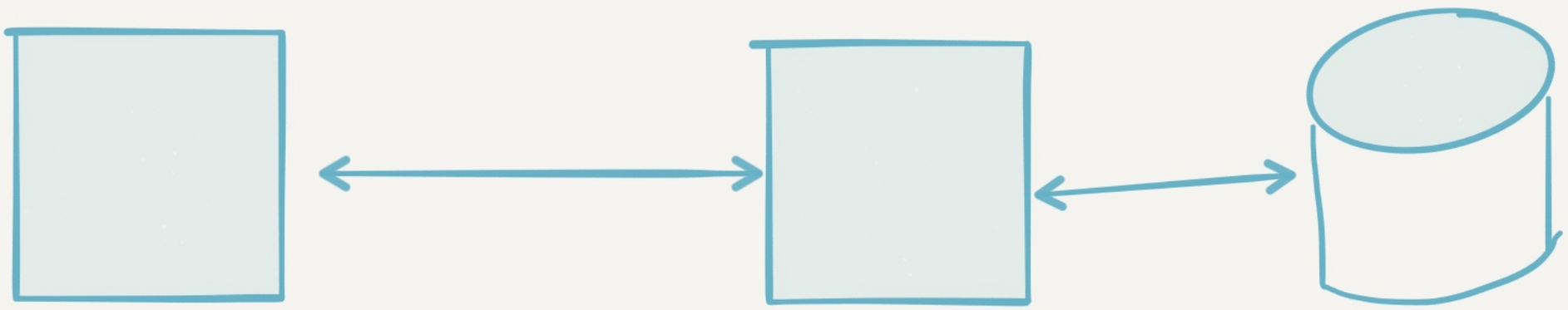
Denise Jacobs, Øredev 2013: When an idea is ready, it doesn't come to just one person.



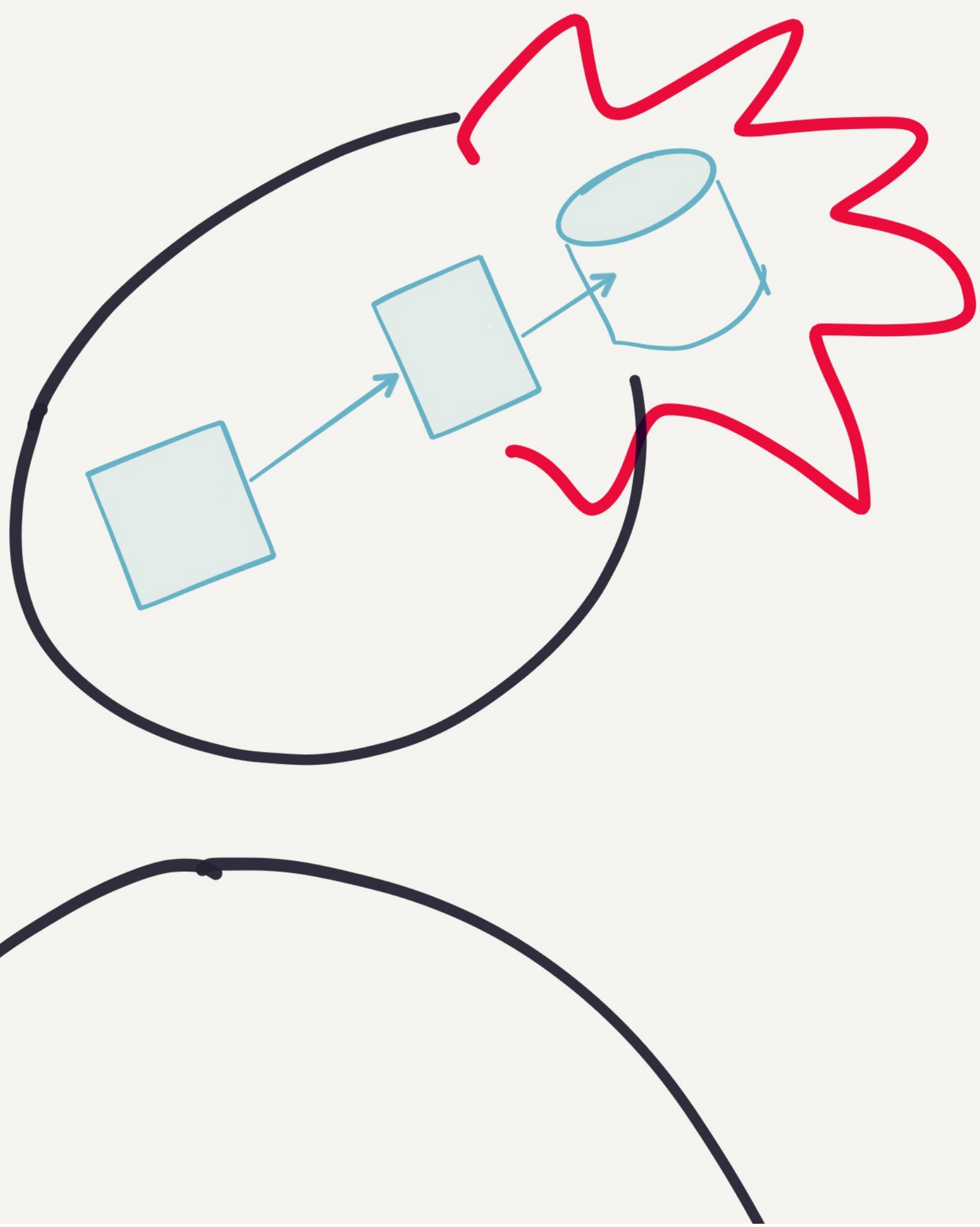
Elixir isn't alone. The actor model, as embodied by Erlang, is also in Akka for Scala and Java. And it is in Orleans for .NET.

Why now? There must be an anomaly, an unsolved problem, in our field.

Distributed systems: they're ugly, they're hard, and they're here to stay. - Camille Fournier

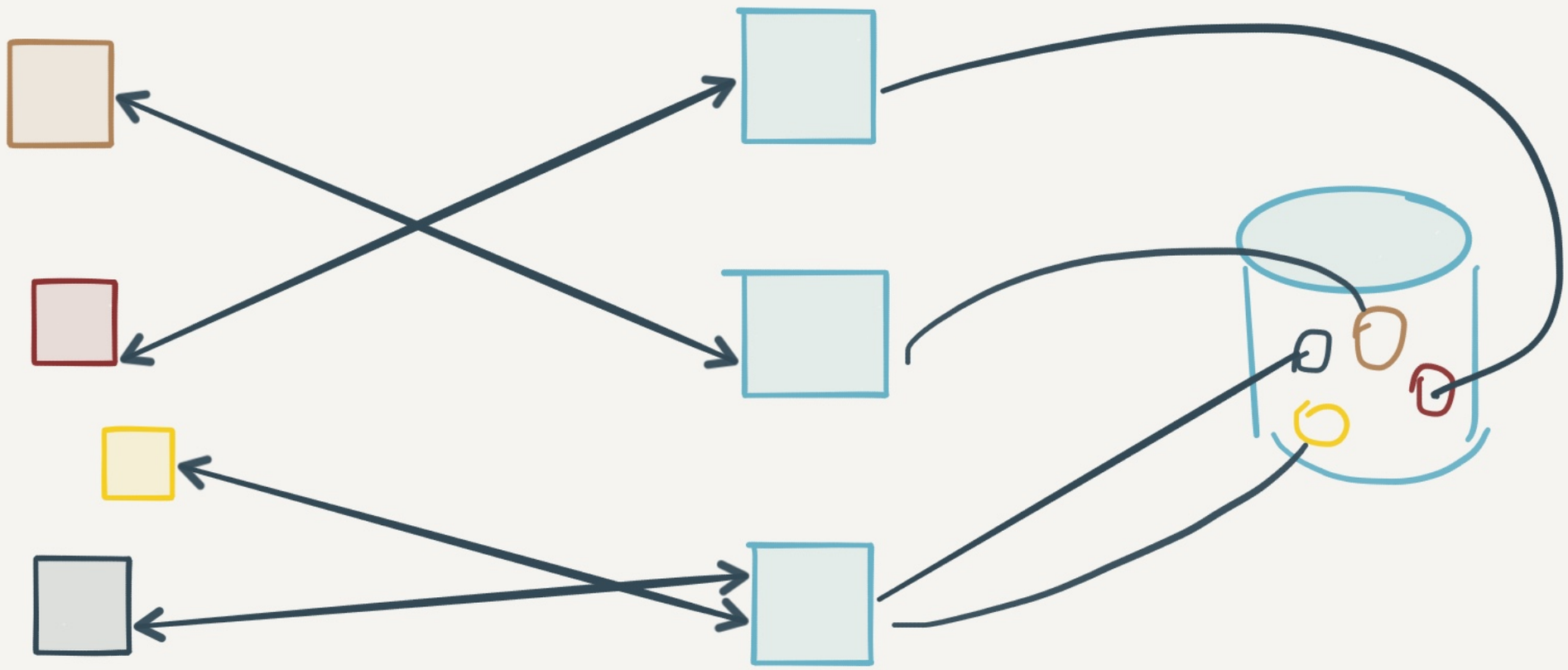


Client-server. It used to be every app. It is still a lot of small apps. It is a distributed system, but not such a resilient one.



Even if you don't have a lot of requests, or a lot of data, there is one scaling problem every program has: scaling to fit in our heads. As features grow, complexity grows combinatorically. The intellectual

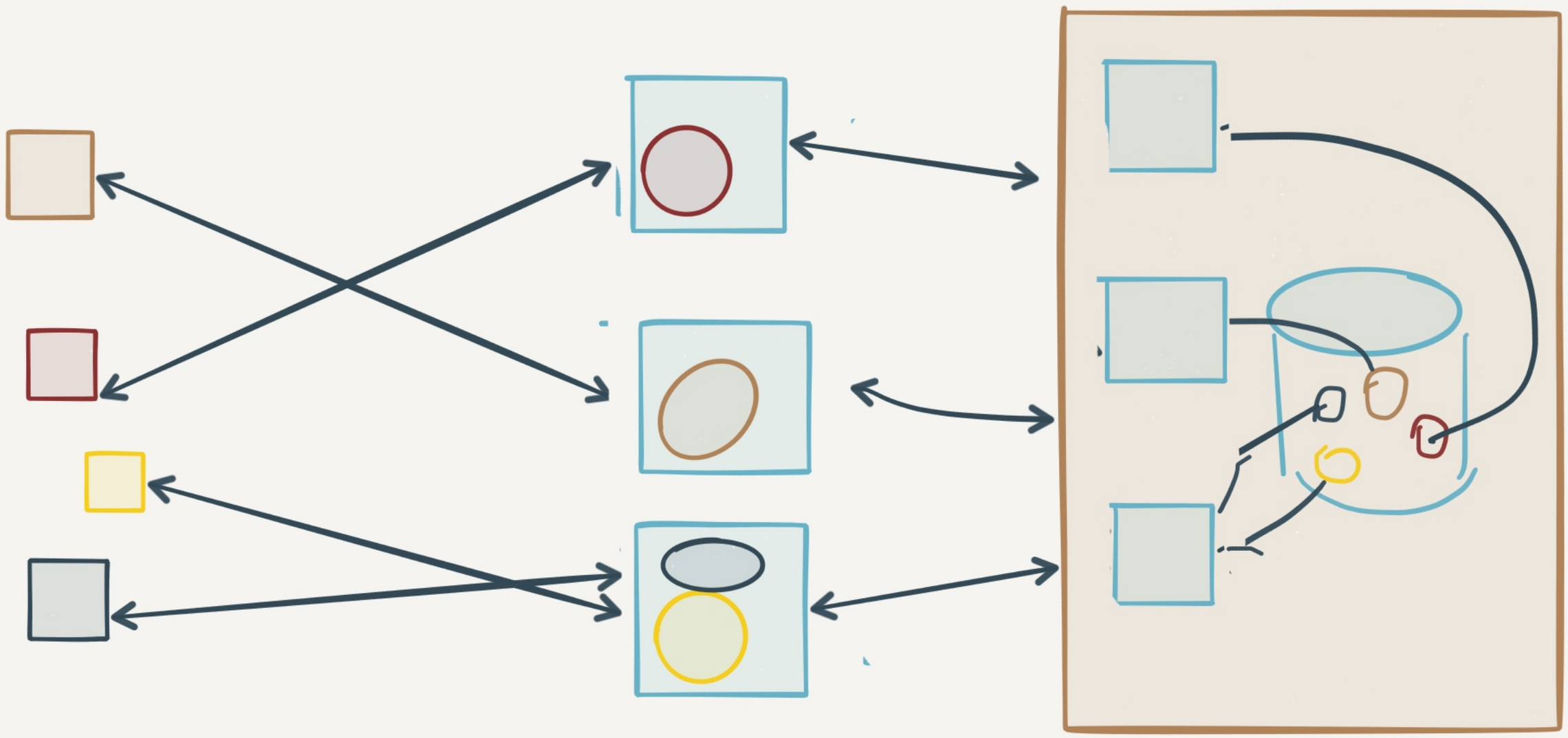
burden is crippling.



Capacity scaling. Horizontal scaling works if the servers are stateless, but then when all the state is in the database (or on the client session), that becomes a bottleneck. We can't be as responsive as we want to be.

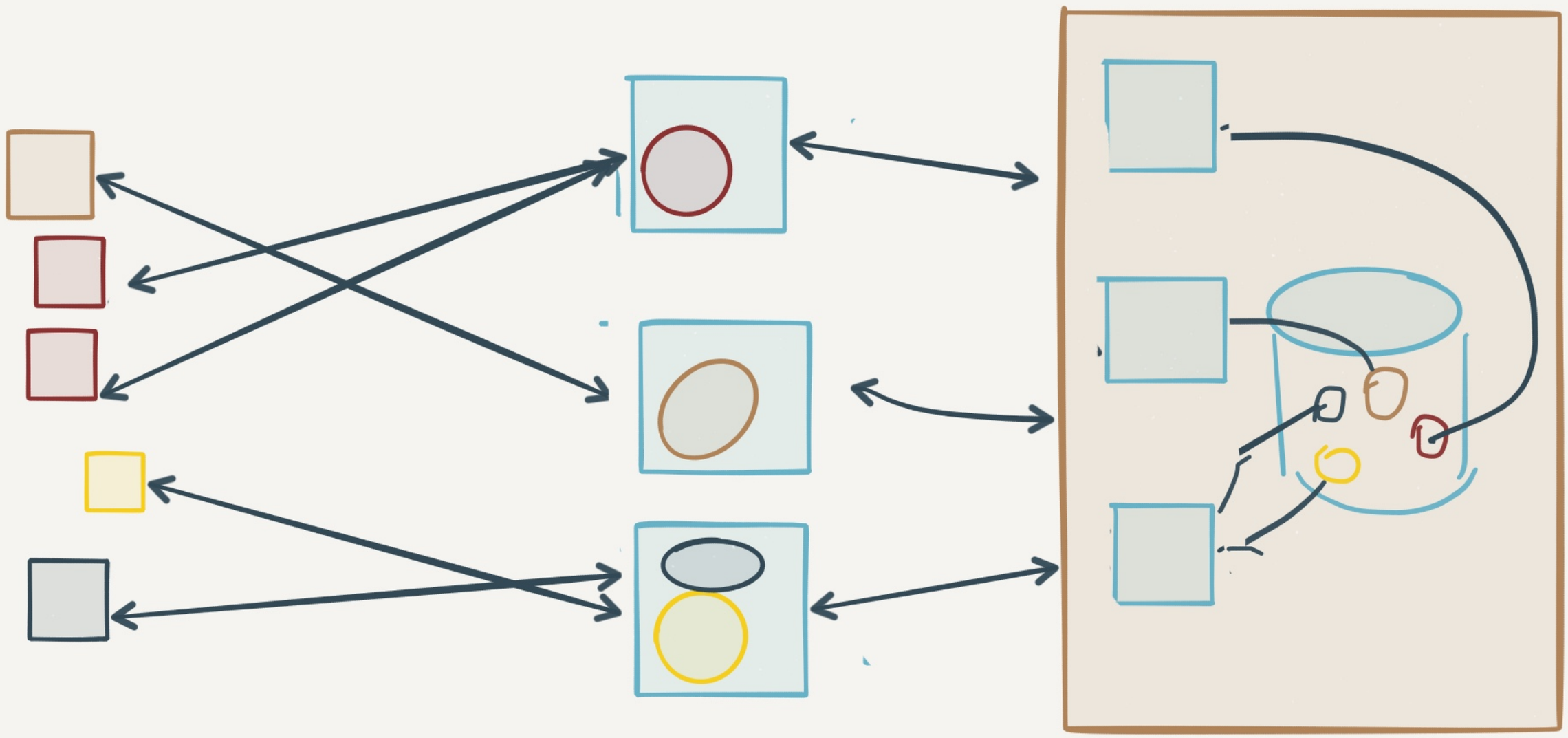
Stateless services aren't efficient enough anymore. Stateful services may be harder, but they're also useful. There are ways to do them right.

Caitie talked about this at StrangeLoop, including Orleans.



With stateful services, a client has a friend on the server, a persistent connection. They can have a rich conversation. Persistent connections are getting to be a thing. Then we push the database into a service, and

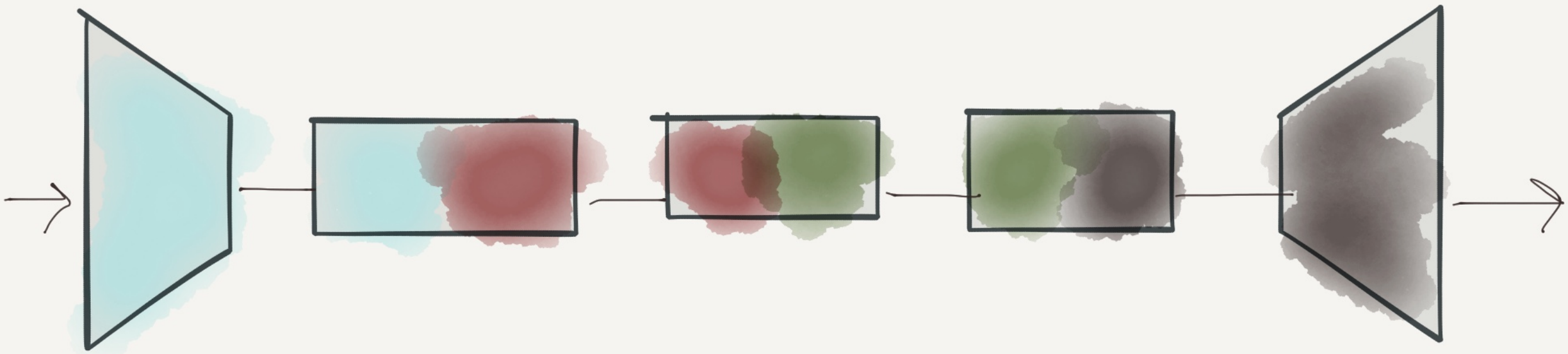
work on scaling that horizontally.



This potentially lets us support multiple clients with the same... Editor session, chat room, shared white board, or to-do list. Not all state belongs in the database, and some of that belongs to more than one client at the

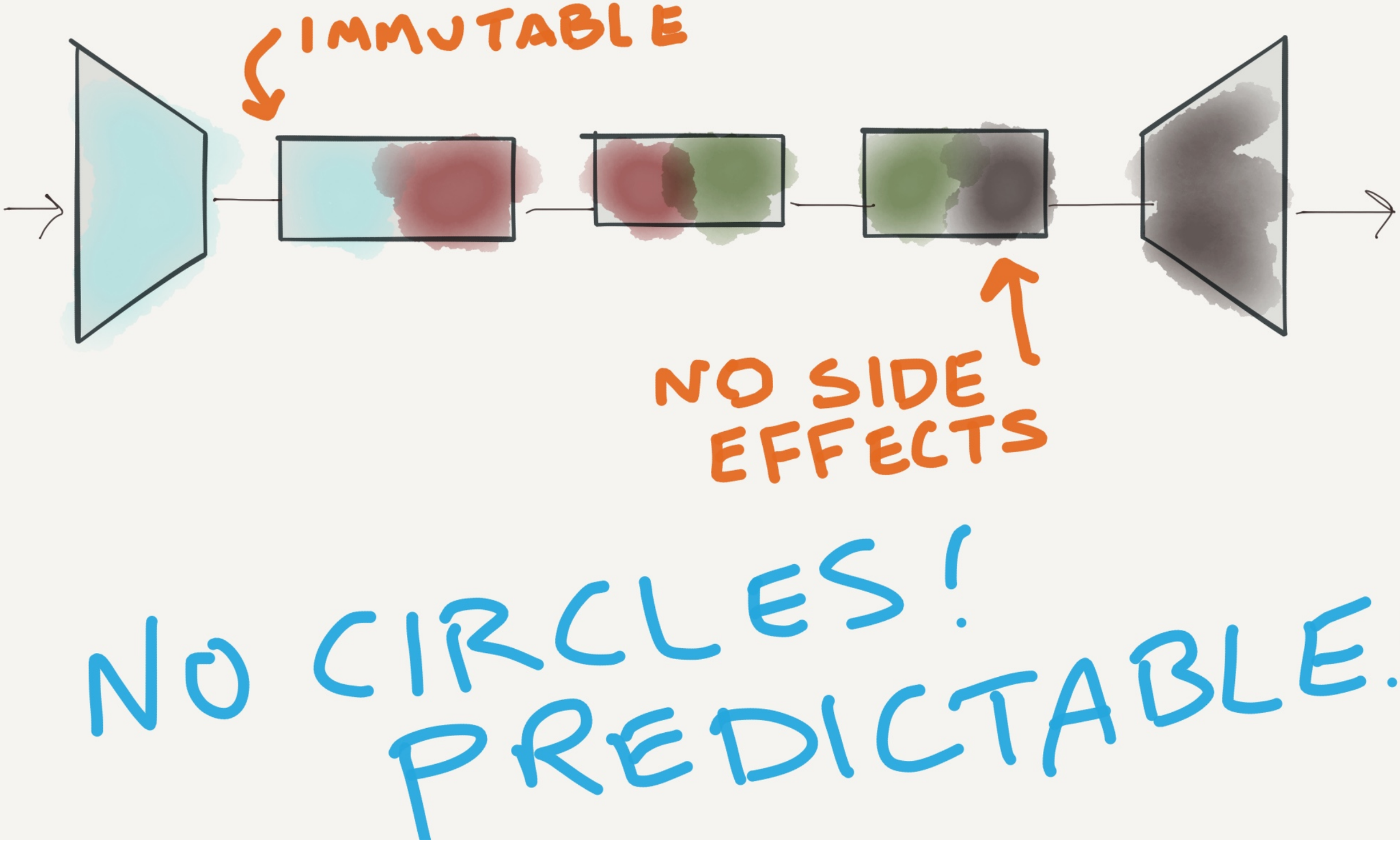
same time. Actor models can work with this.

Surely we can solve this with FP, right? It's all the hotness!

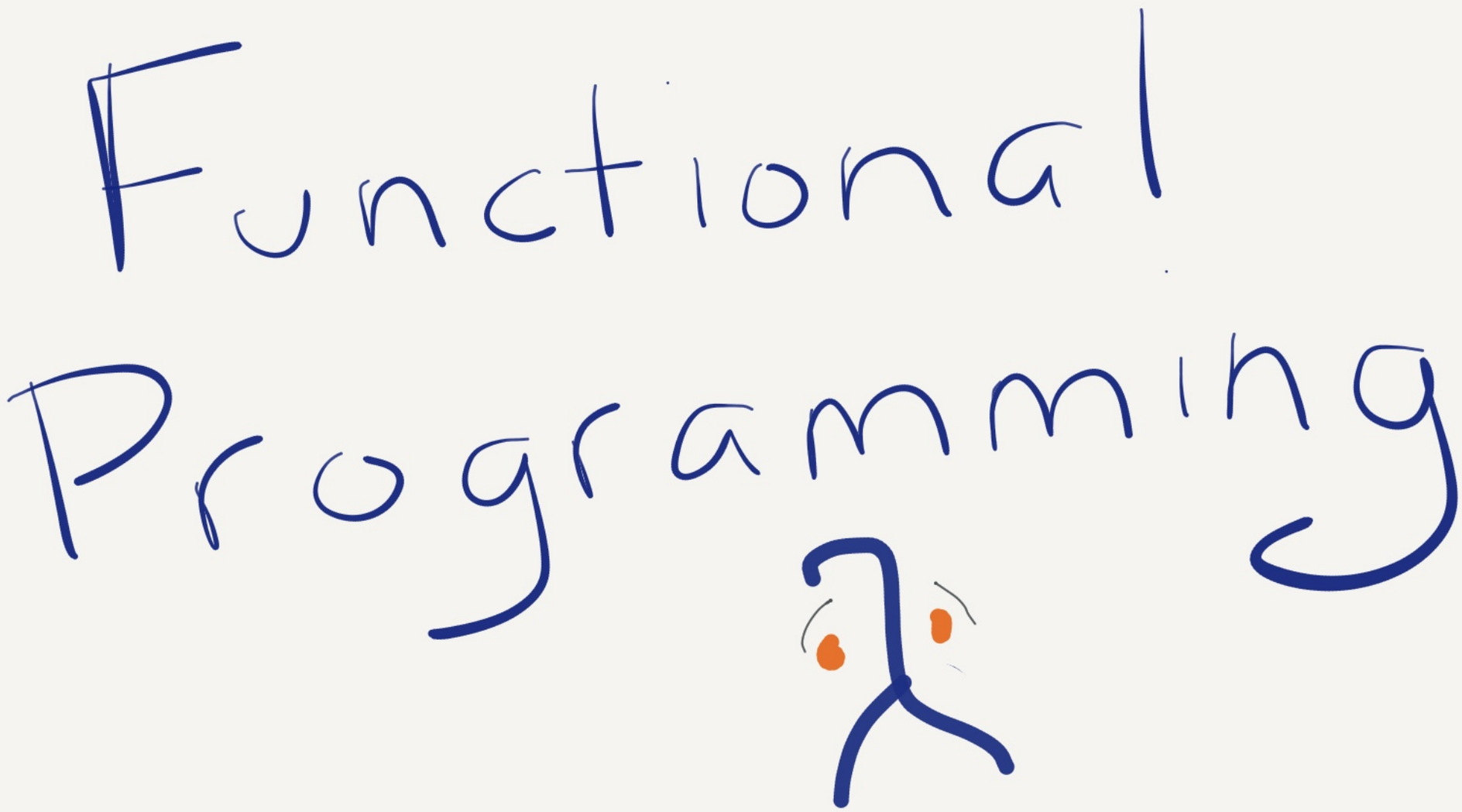


Flow of Data

Functional programming, at its essence, is about a flow of data. Data in, data out, highly testable. This works for request-response services.



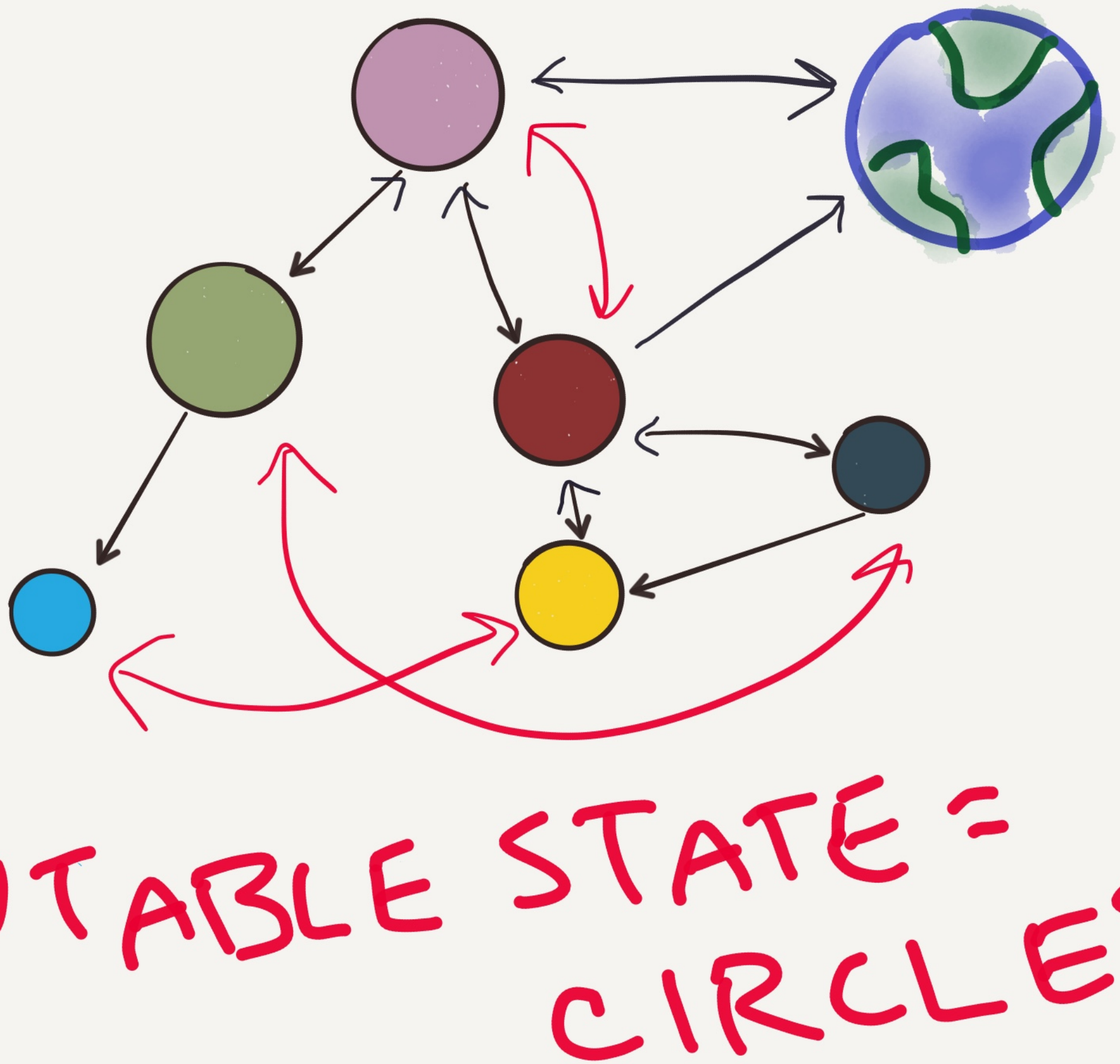
The beauty of this is its predictability. For the same input, always the same output. Testable, fits in our heads.



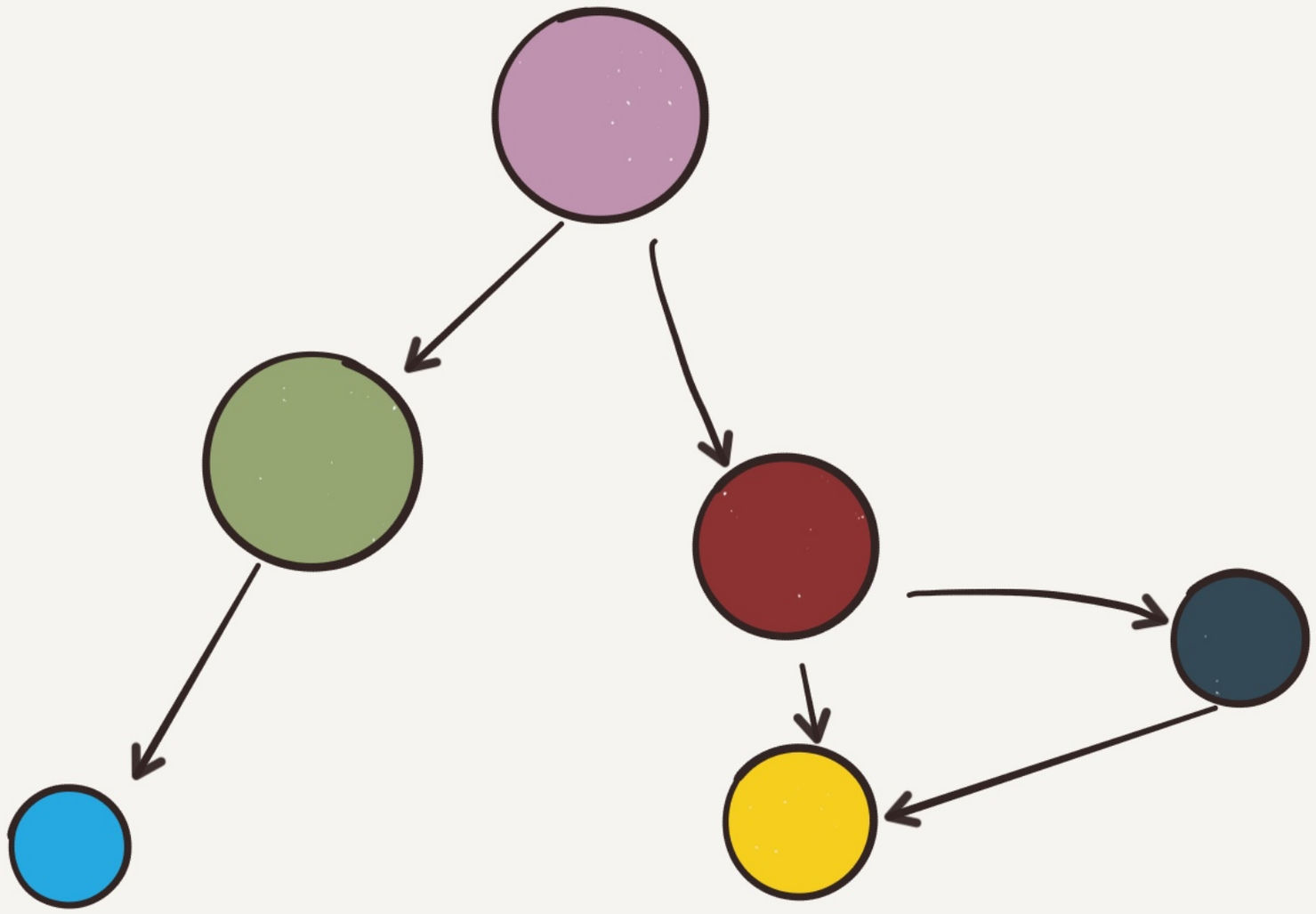
Unfortunately, functional programming is limited. Lambda calculus does not support concurrency.

is not enough

Object oriented programming, also not enough. Our big OO apps don't fit in our heads.



Our classes are all in a snarl. Nothing stops us from lots of circular dependencies. Plus mutable state and side effects are a bunch of invisible circular dependencies.

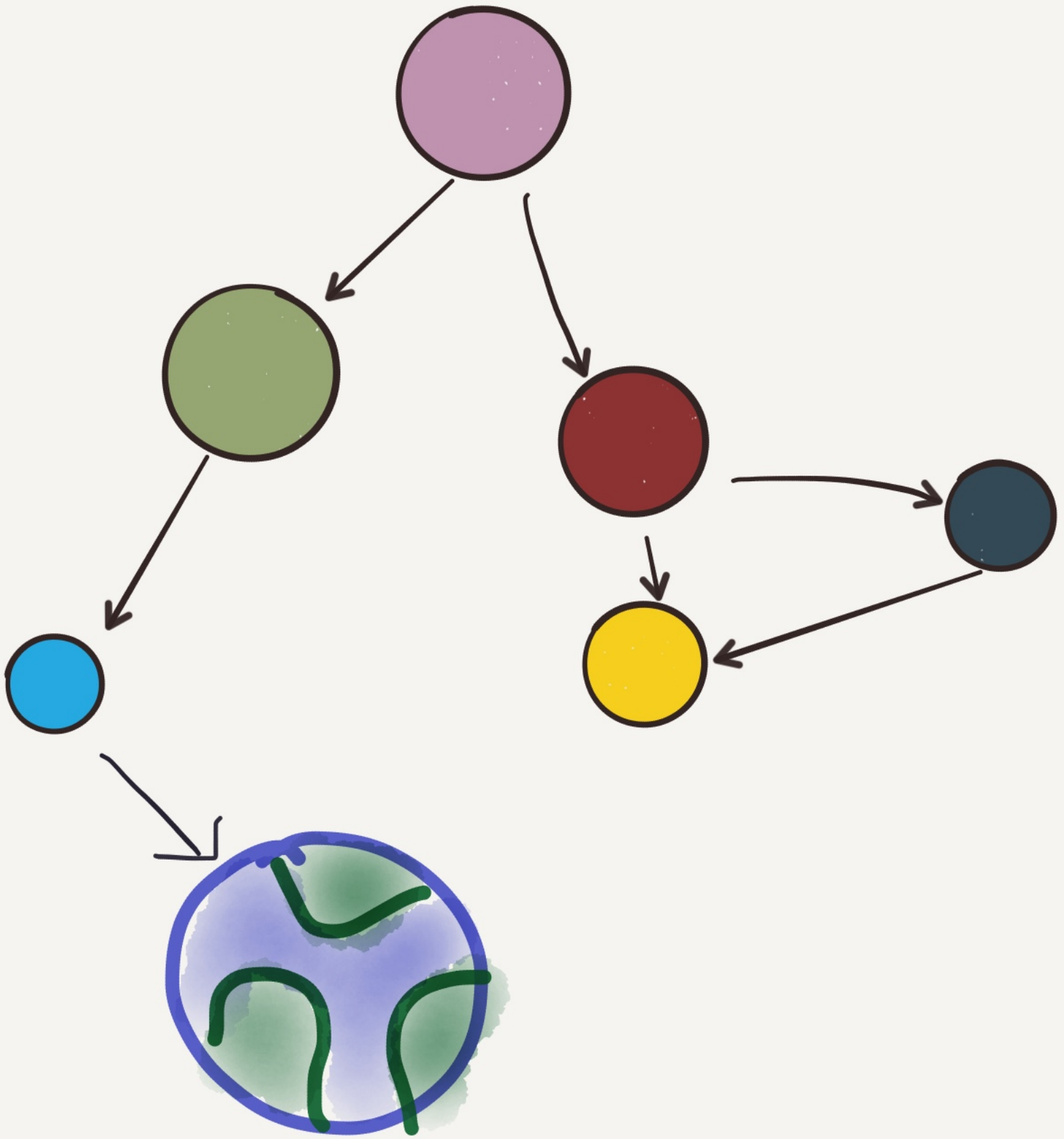


Tell Don't Ask

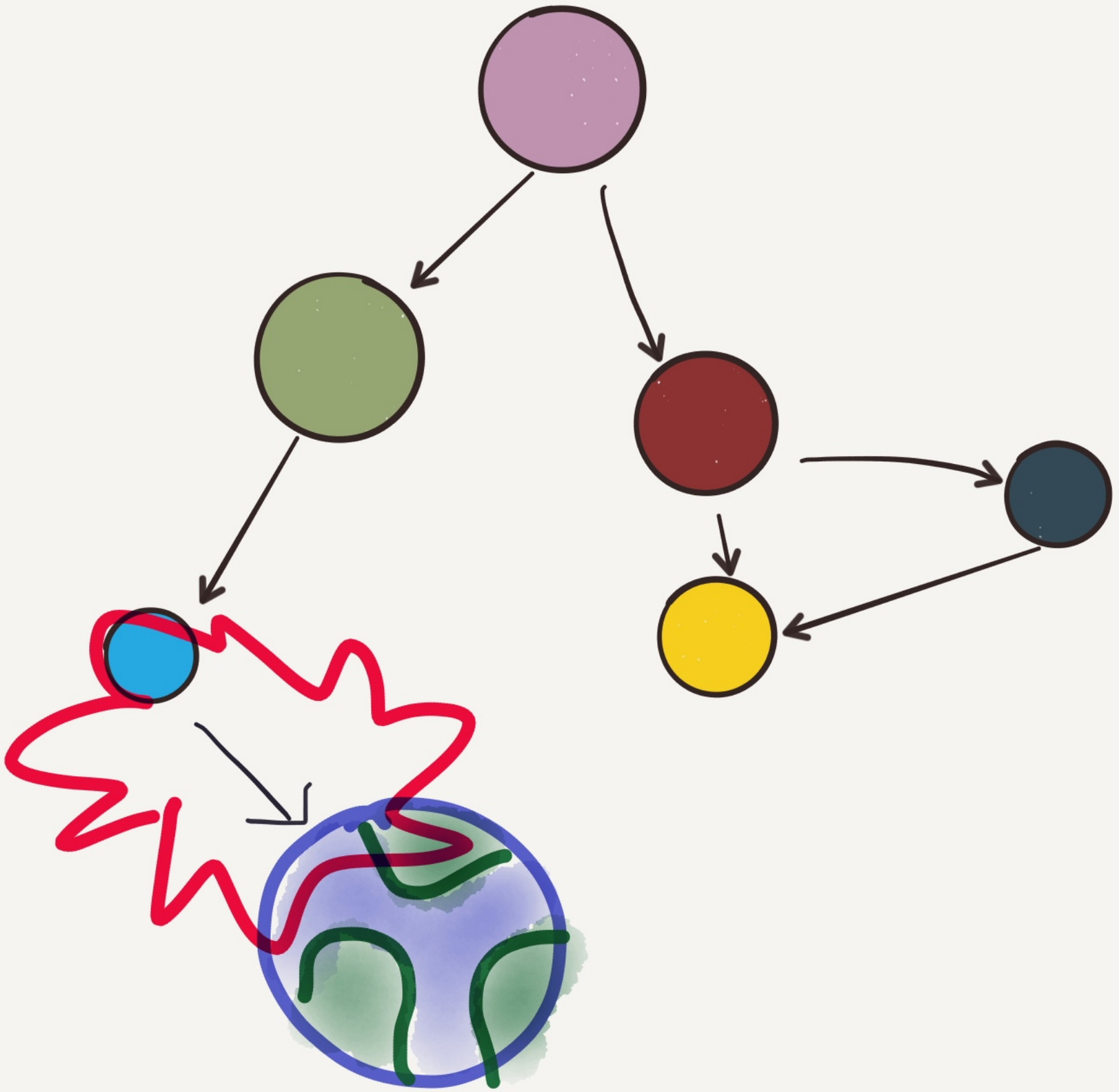
OO, at its roots, according to Alan Kay, was about Tell Don't Ask. Asynchronous messaging.

We've gotten away from that with dependency injection and getters and setters. We can choose to make our classes have an acyclic structure. We can use the organizational strength of OO for grouping code together.

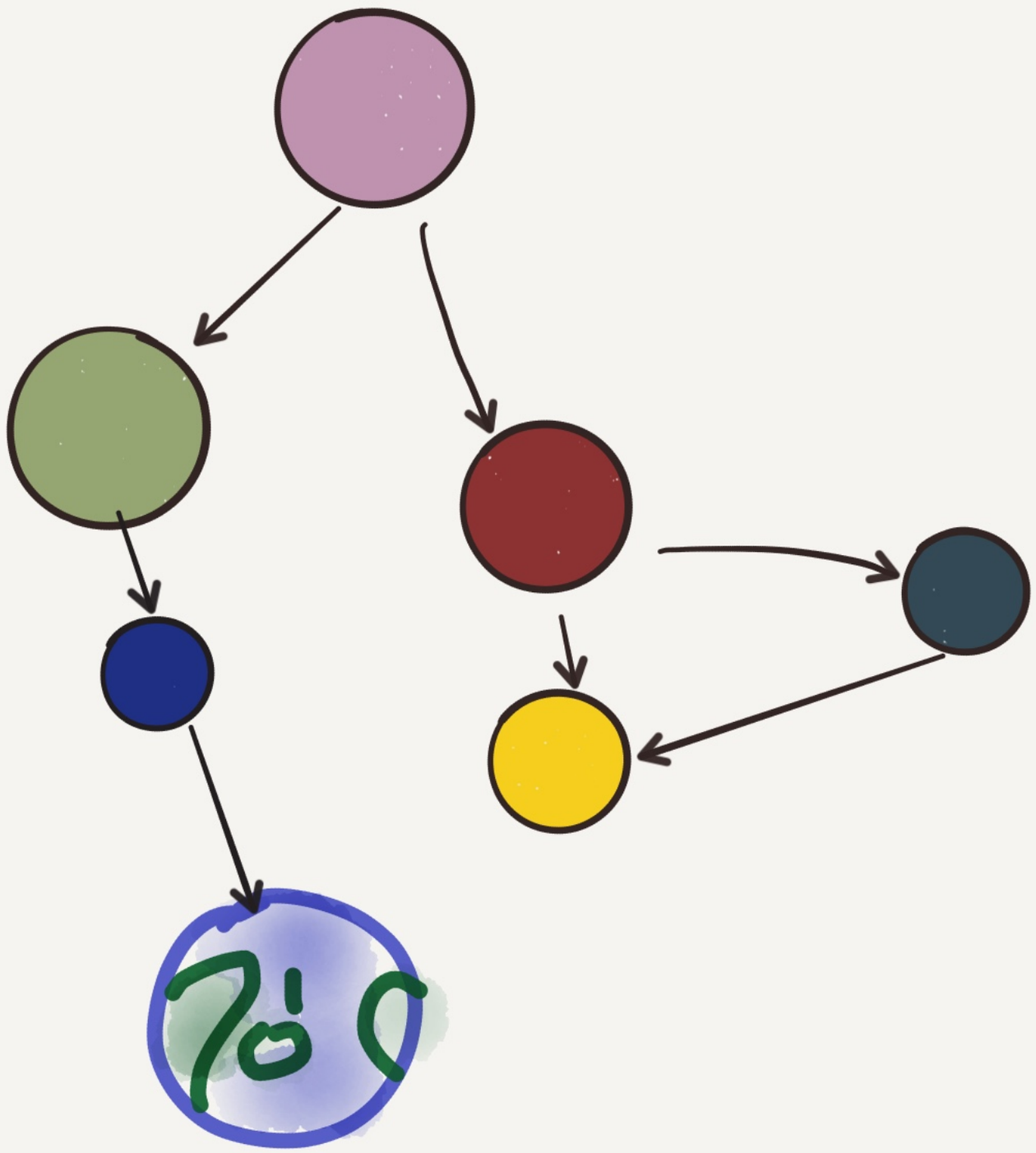
And actors are the epitome of Tell Don't Ask.



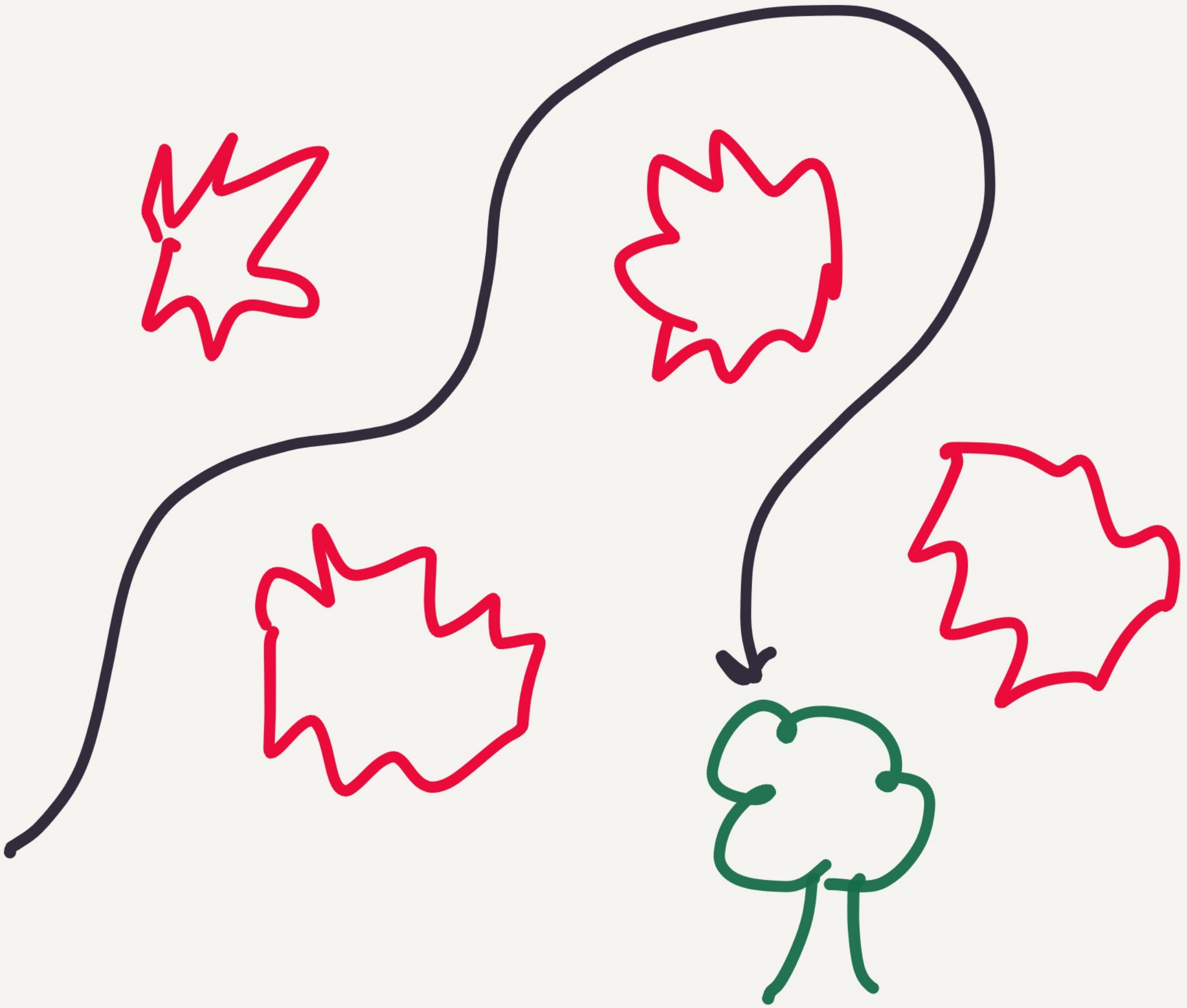
From here we can access the outside world. And it might fail. It might take our actor with it.



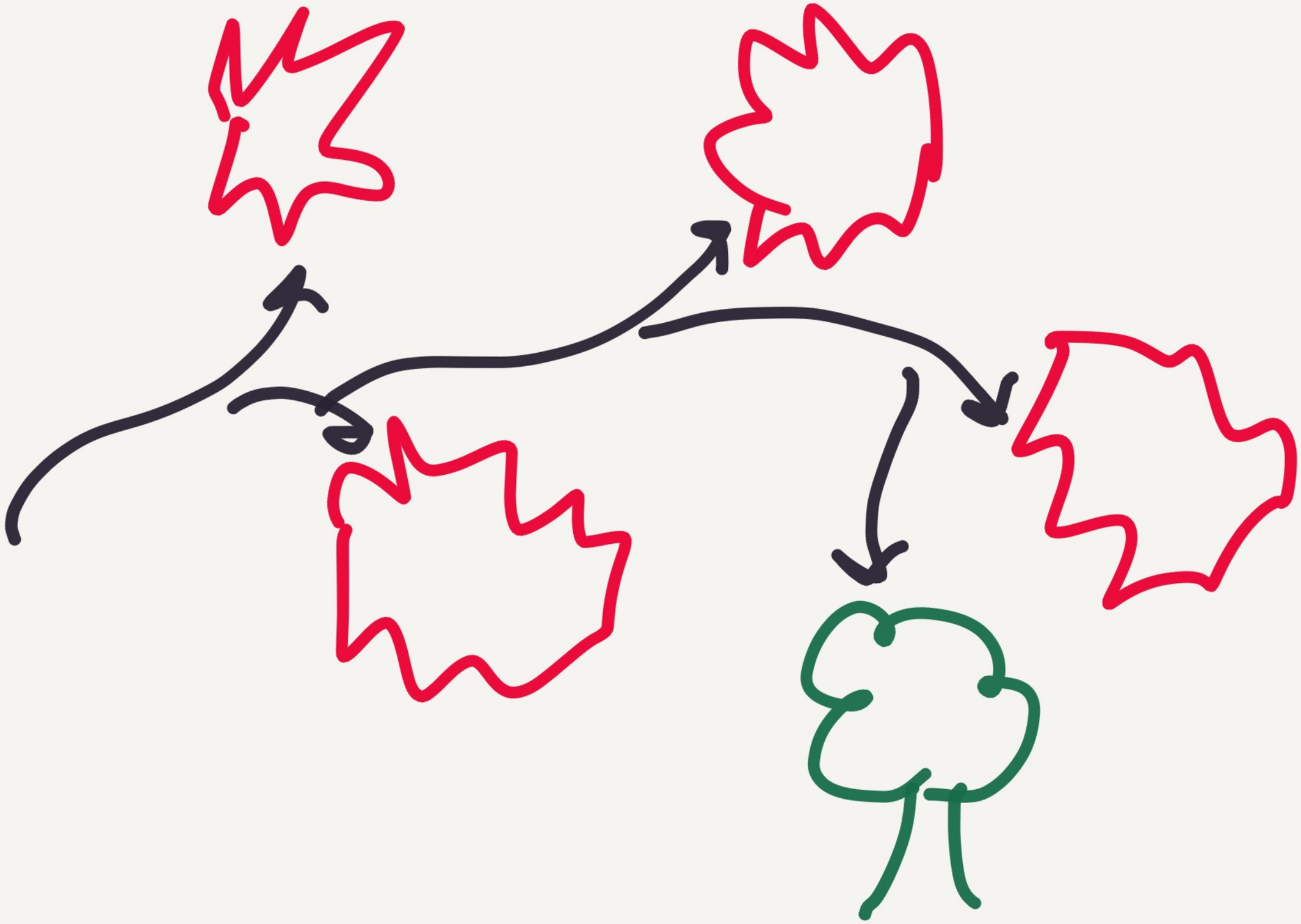
No bother! We'll just spin up another one.



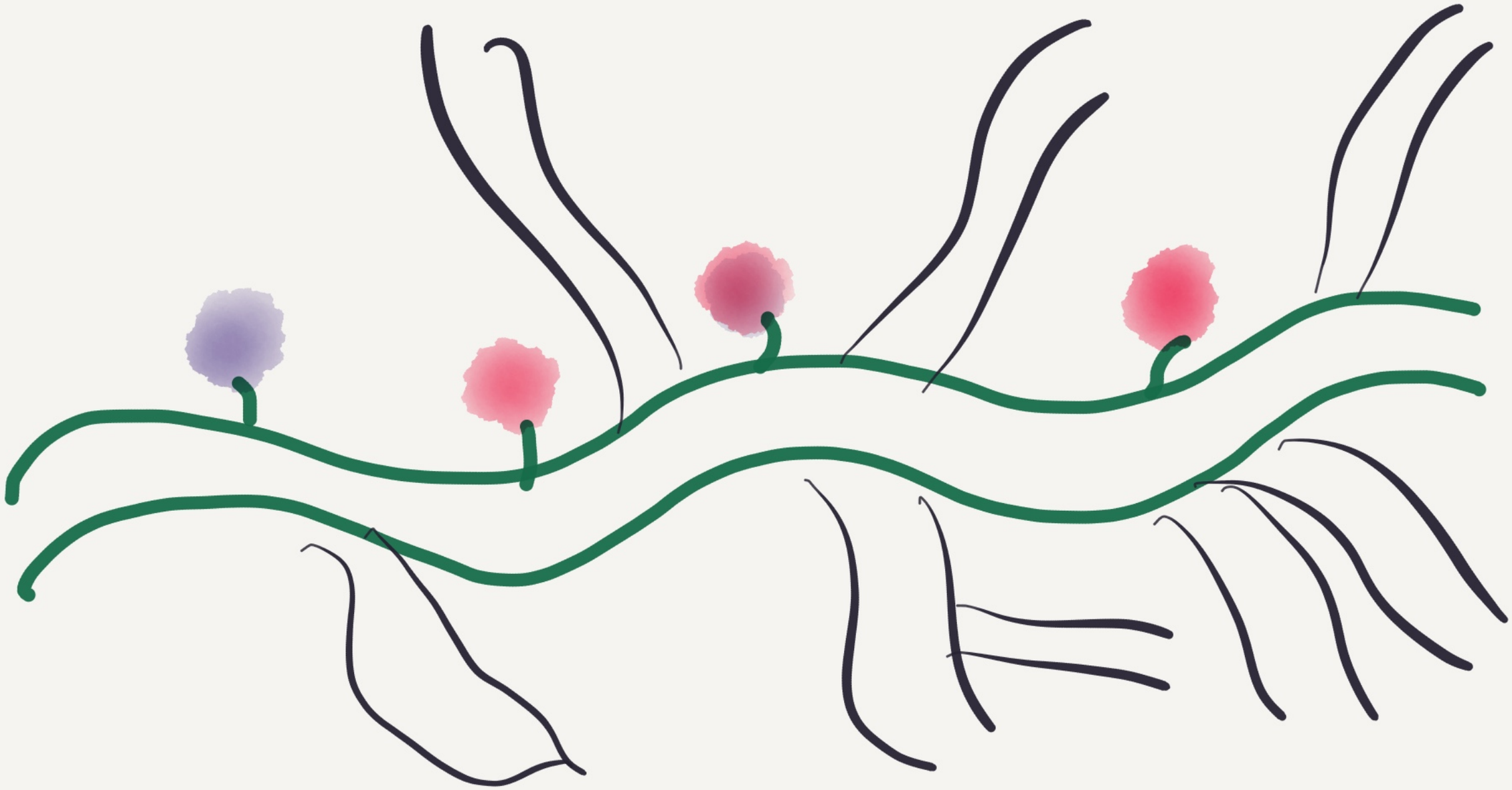
We can make a new actor, and it can connect to a new world. Bruce Tate said this morning, concurrency without isolation is just noise.



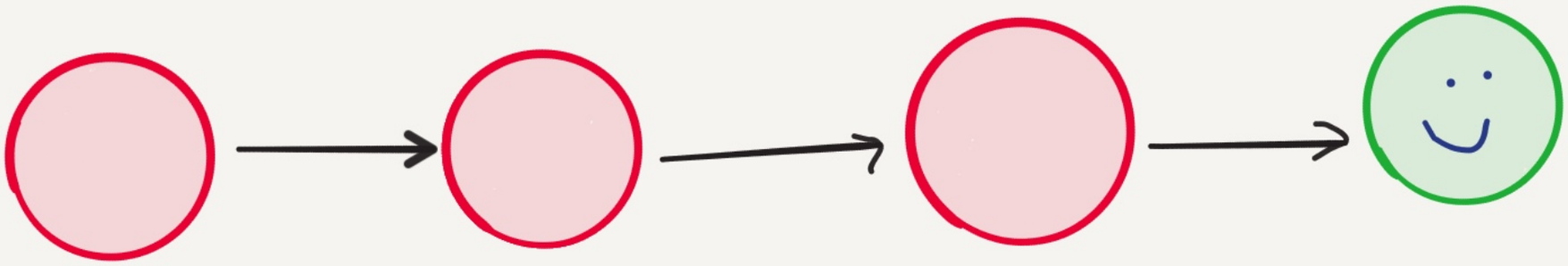
Up until a few years ago, the only cultural narrative was: success comes from avoiding failure

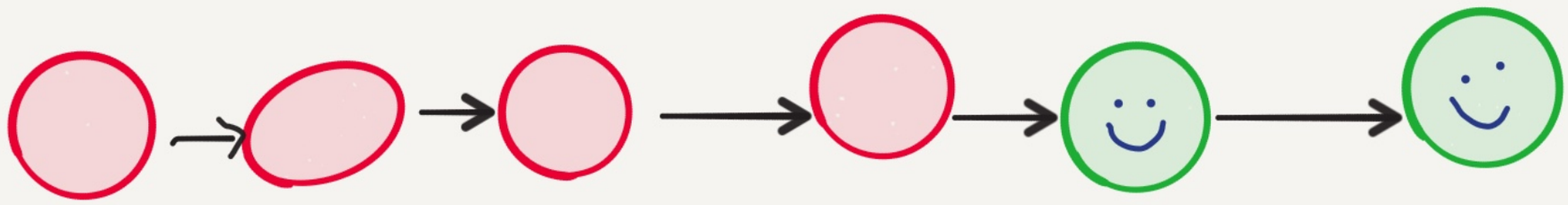


But it's more like, success comes from lots of failures. Recognizing and learning from failure. That's how we find the path. Failure is not an option: it is inevitable.



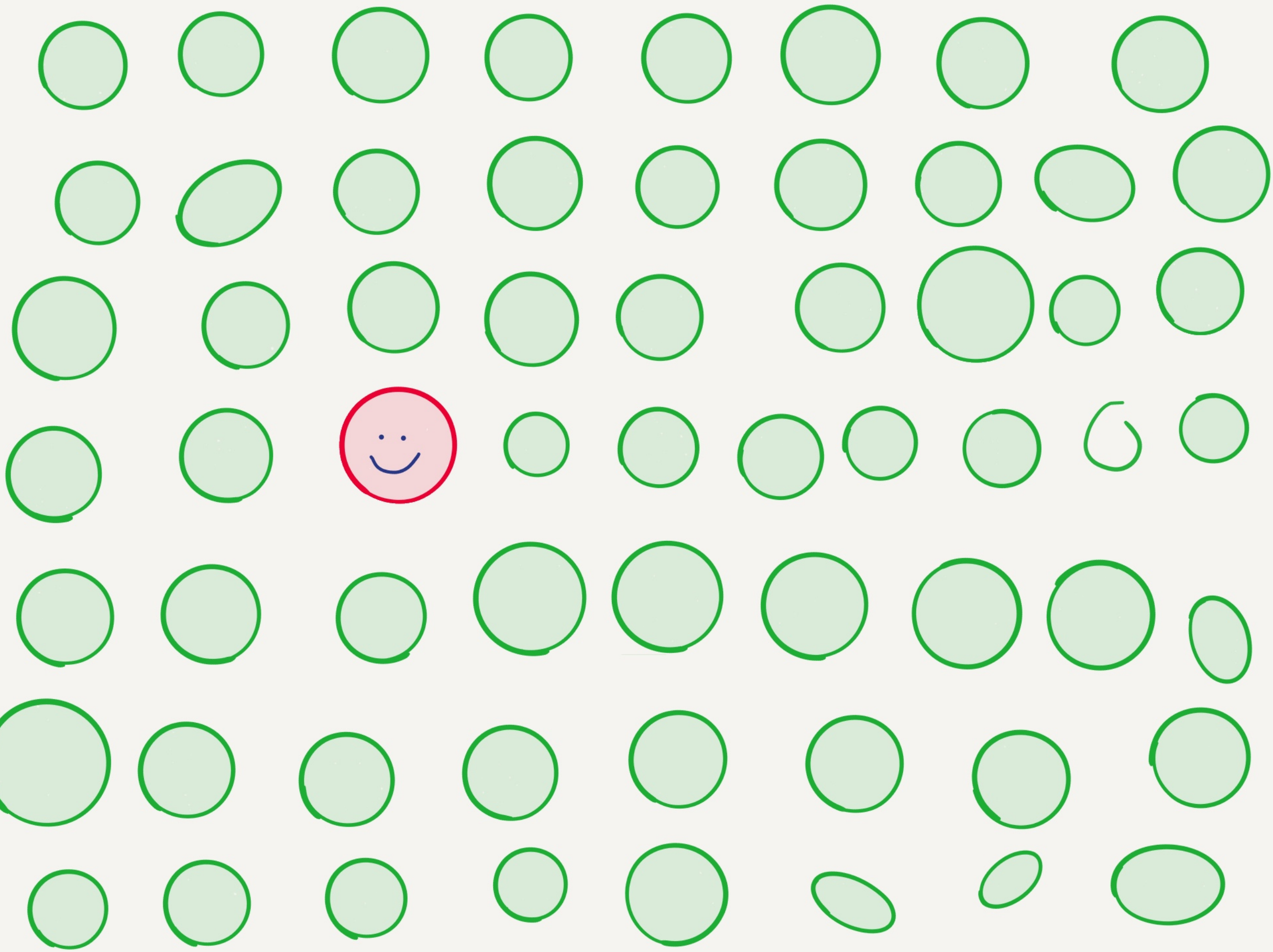
The happy path is one path. Maybe a few. The errors paths are nearly infinite! Count not the quantity of travelers, but the quantity of paths. Realize that most of your code should be error handling code.





Tests: red, work. Red, work. Red, work. Green, stop.

In testing, the common case is failure.



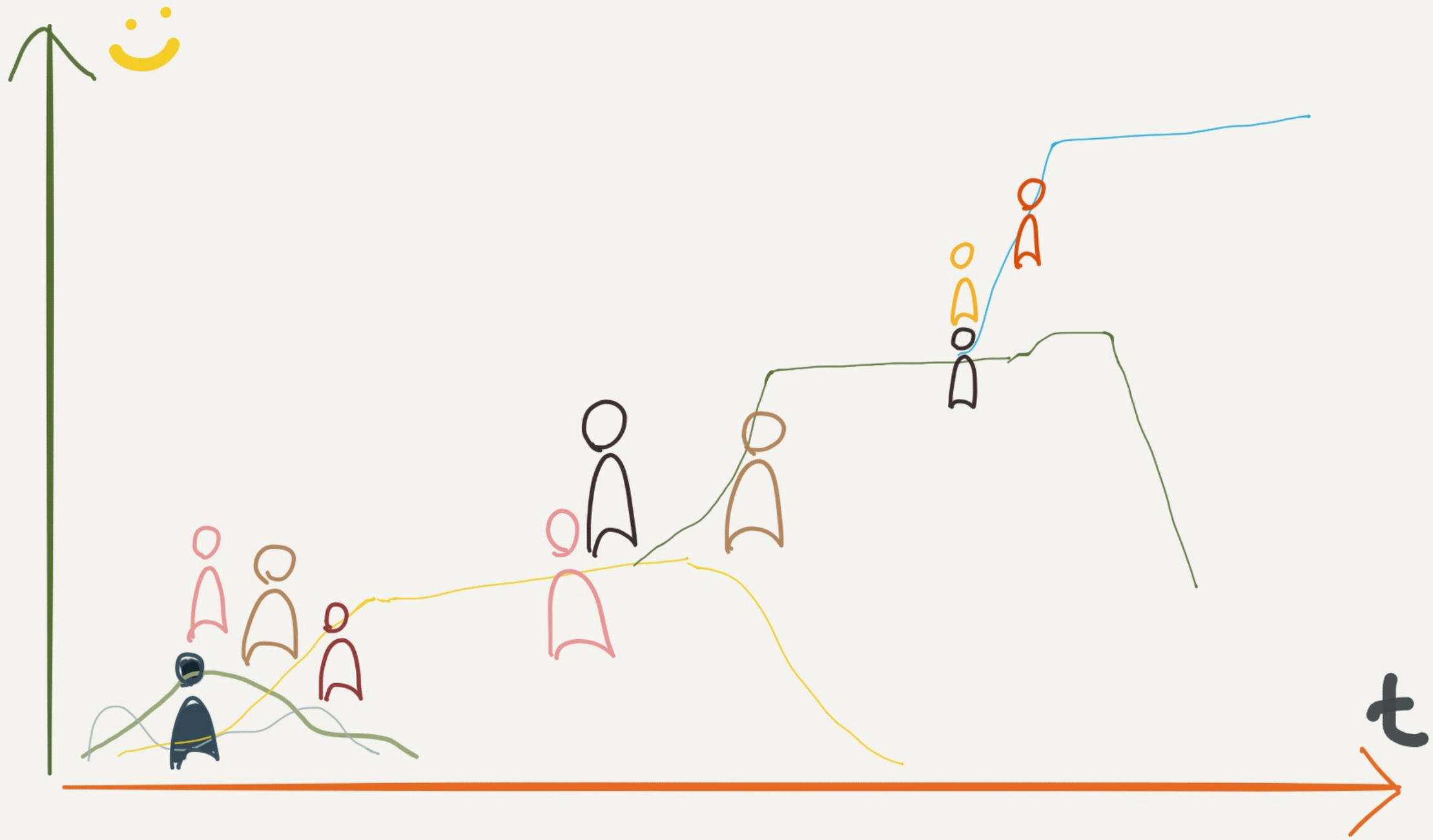
Property tests are great. They help us find the one failure among a lot of green, without having to write a ton of test cases.

Erlang has great tooling for property tests. See: QuickCheck CI. Elixir could make use of this.

FAILURE IS THE COMMON CASE

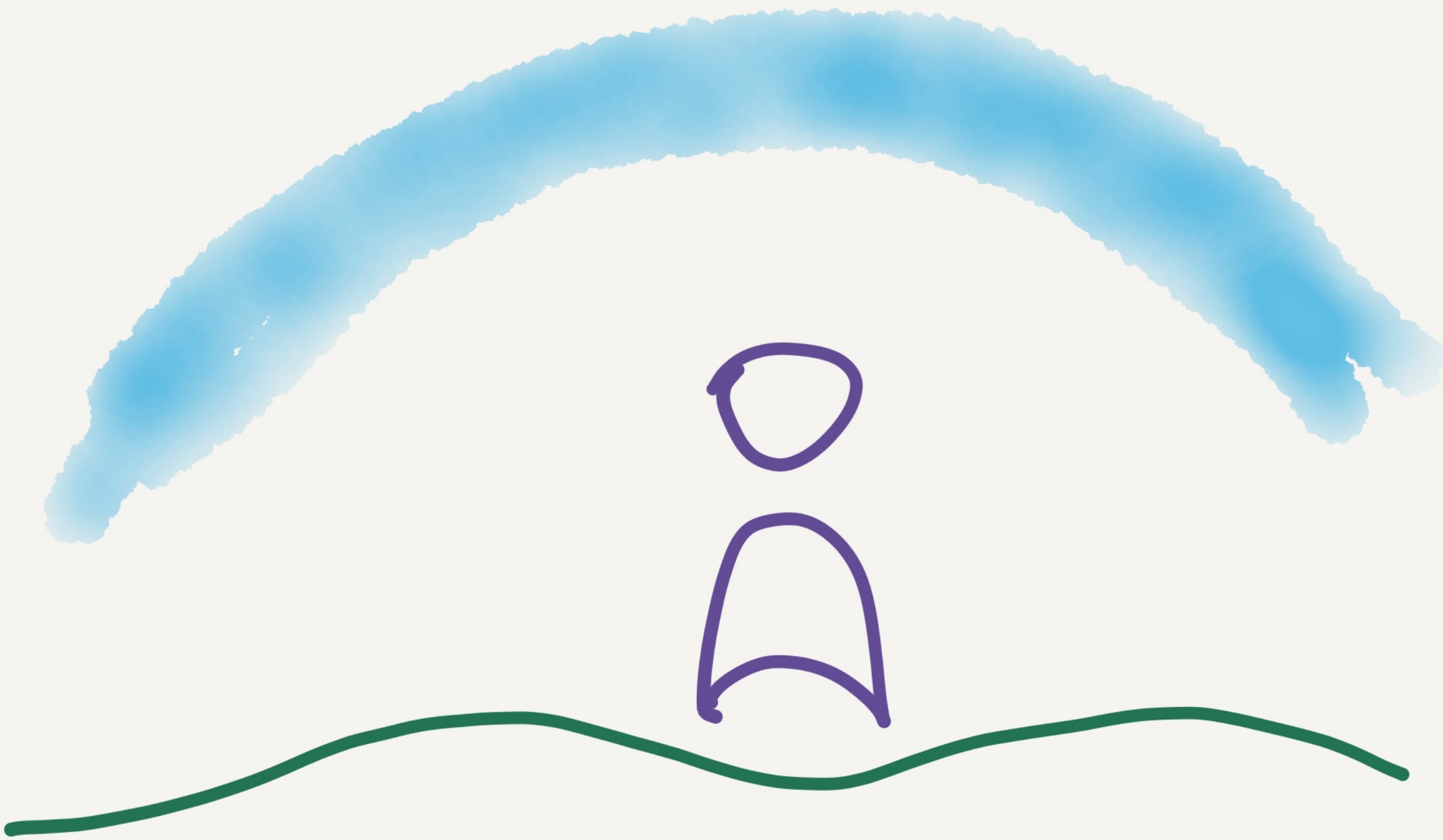
Accept it. Embrace it. Code for it.

At least Elixir gives us supervisor hierarchies, ways to deal with unanticipated failure separated from our happy path code.



Back to science. To ideas. Failure is the common case here, too, if failure is a Wikipedia article declaring you the 3rd greatest physicist of all time.

But if success is contributing, is promoting and participating in a community that enjoys this useful paradigm that is the Elixir programming system, then success is varied and common. A community defines a paradigm, it carries forward the methods and values of it.



So the sky is the limit, right? We can go all kinds of places with Elixir.

Yes, the sky is the limit. And that's a problem, because there is something beyond that. What happens when we break through the sky?

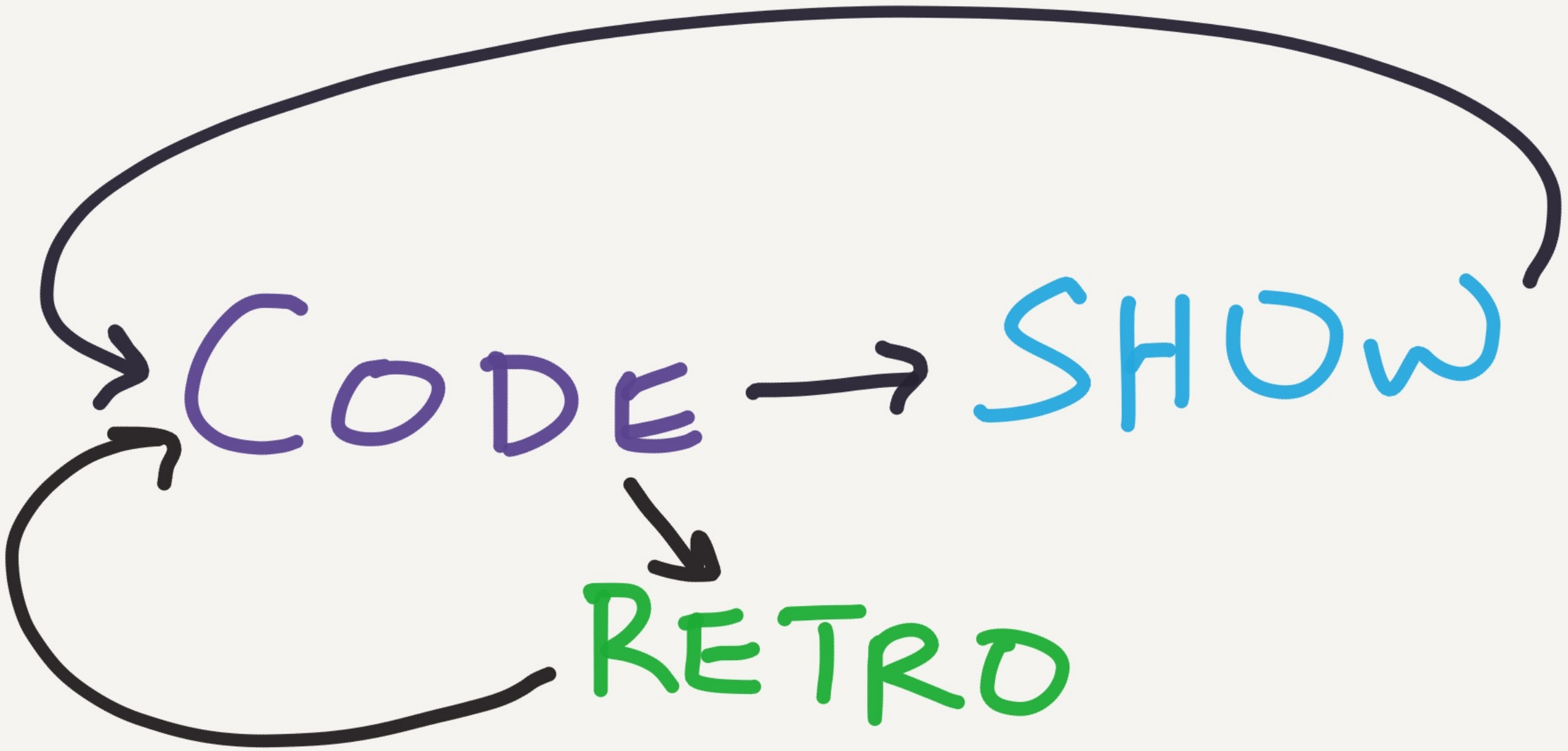
Ideas, they keep coming. After we think we are done, we have figured it out, new people come in, and they have whole new ideas.

In science sometimes it takes a generation for these new ideas to become accepted, as people cling to the old. To ether. To electricity as a fluid.

We can do better. Ideas flow quickly in our young field. We can stay open to them. We can think, "man I love this way of working! I wonder what will be even better."

What Comes After Agile?

Answer: Lean.

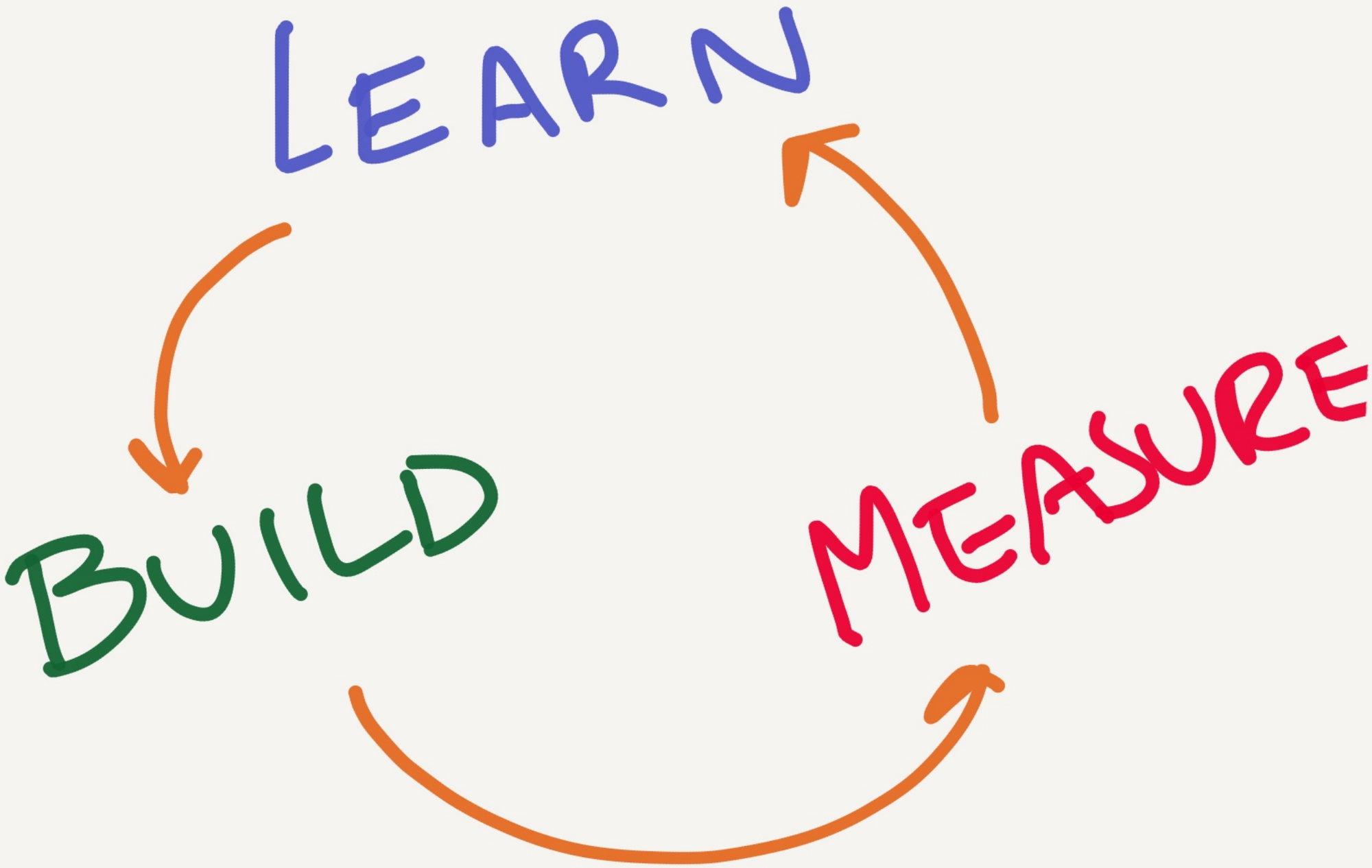


In Agile, we build a bit, show it to the customer, then build more based on that feedback.

We also work, then reflect on our work, then change our working style for the next iteration. See the feedback loops. This is how we learn and grow. This is how we discover.

This is Marty Cagan speaking at Craft Conf in April 2015. He talked about the product cycle on a much higher level than the agile team. Shorten feedback loops in the whole company. Find failing ideas before

building them into software, when we can.

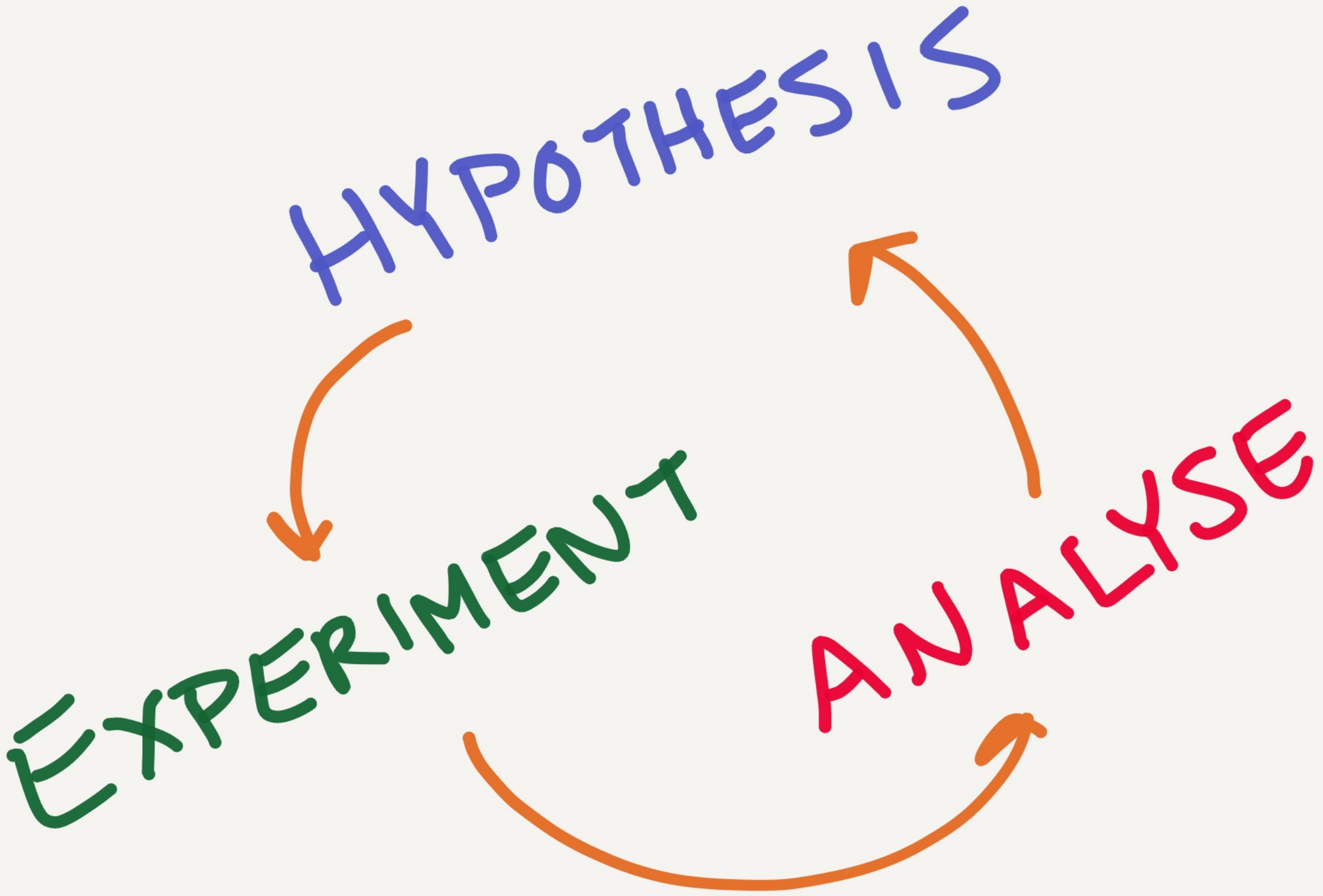


After Agile is pretty much Lean. It's build, measure, learn - emphasis on the learn. It's feedback loops for the business, and it's shorten those

feedback loops as much as we can.

Loops. Circles. These make things interesting. They're where we get something we could not have predicted.

You want these in your team. Not in your code.



Someone pointed out, this looks like the scientific method.

Kuhn teaches that it's rarely so simple. People discard the data they don't

like, ignore results that don't fit with the theory. Ampère ran experiments that revealed induction, but they didn't help prove his theory, so he ignored them.

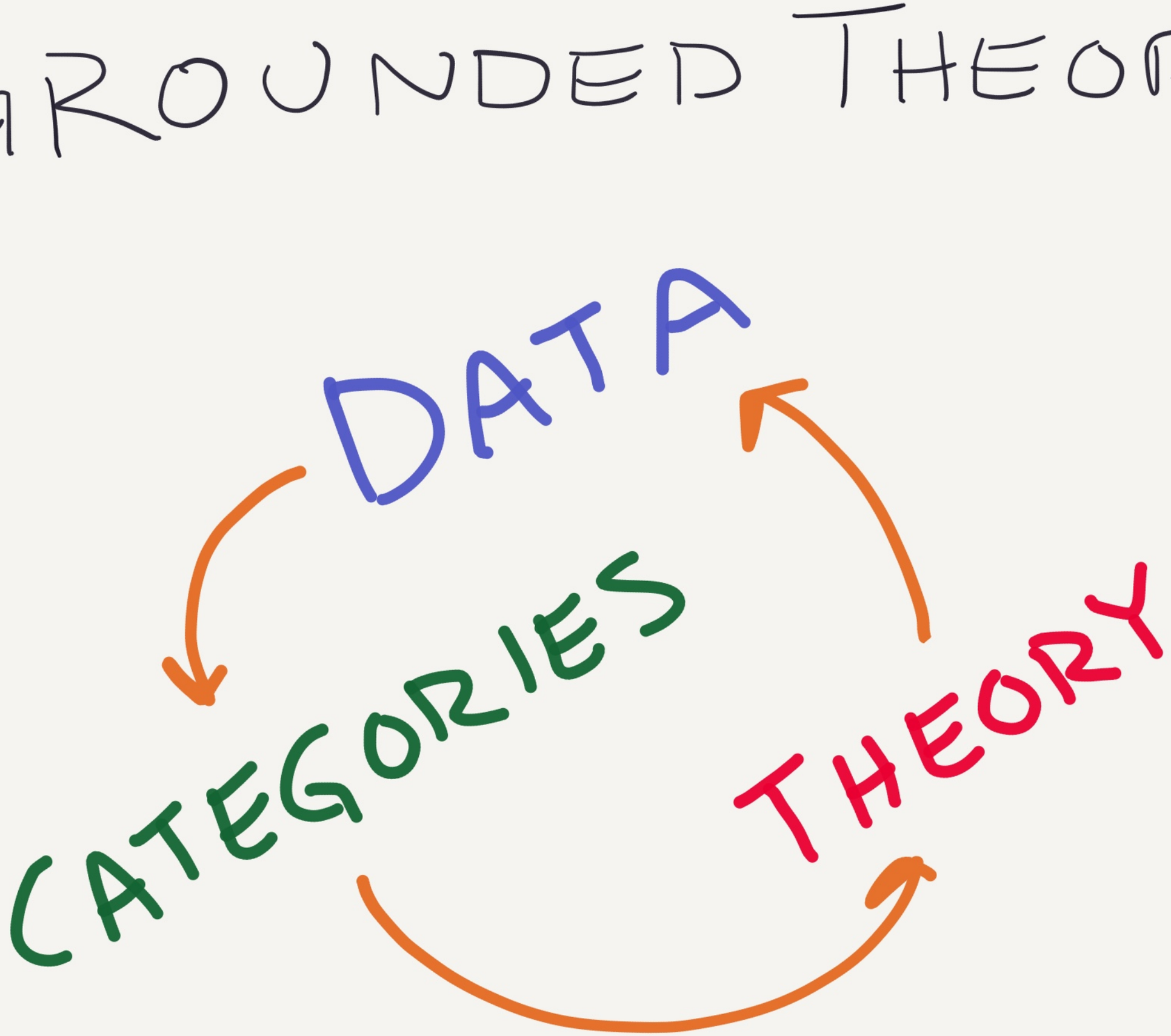
Regardless. Here's the limitation: the scientific method can't help us study systems we are part of. It requires objective, separate observers. Near-identical control groups. Repeatability.

If we want to know how our team could work better, we don't have a control group. We have one team of unique individuals with a particular problem to solve. How can we study human systems from the inside?

What Comes After The Scientific Method?

Answer: grounded theory

Brené Brown is a social scientist. She has some amazing TED talks. And a book, the most exciting chapter of which, for me, was the appendix. There she talked about the research methods she used.



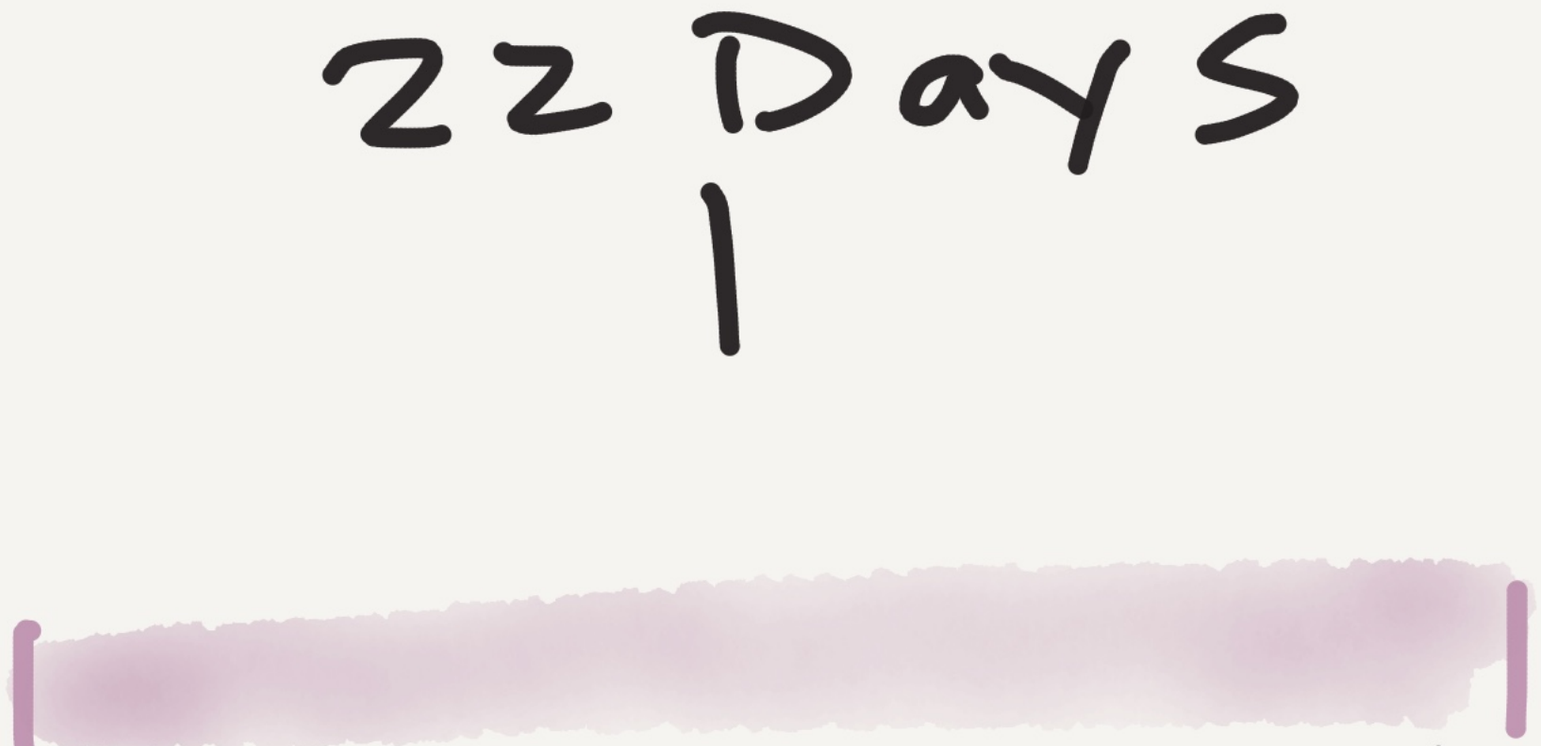
Grounded theory starts with a question. "What is wholeheartedness?" Or "What are constructive team interactions?" Then gather data, mostly stories. Then categorize the parts of the stories. Categories lead to

theories. Theories lead to more questions. That leads to more data gathering. Study the system from within.

What Comes After

NO ESTIMATES?

Answer: ranged estimates.



This comes from Dan North.

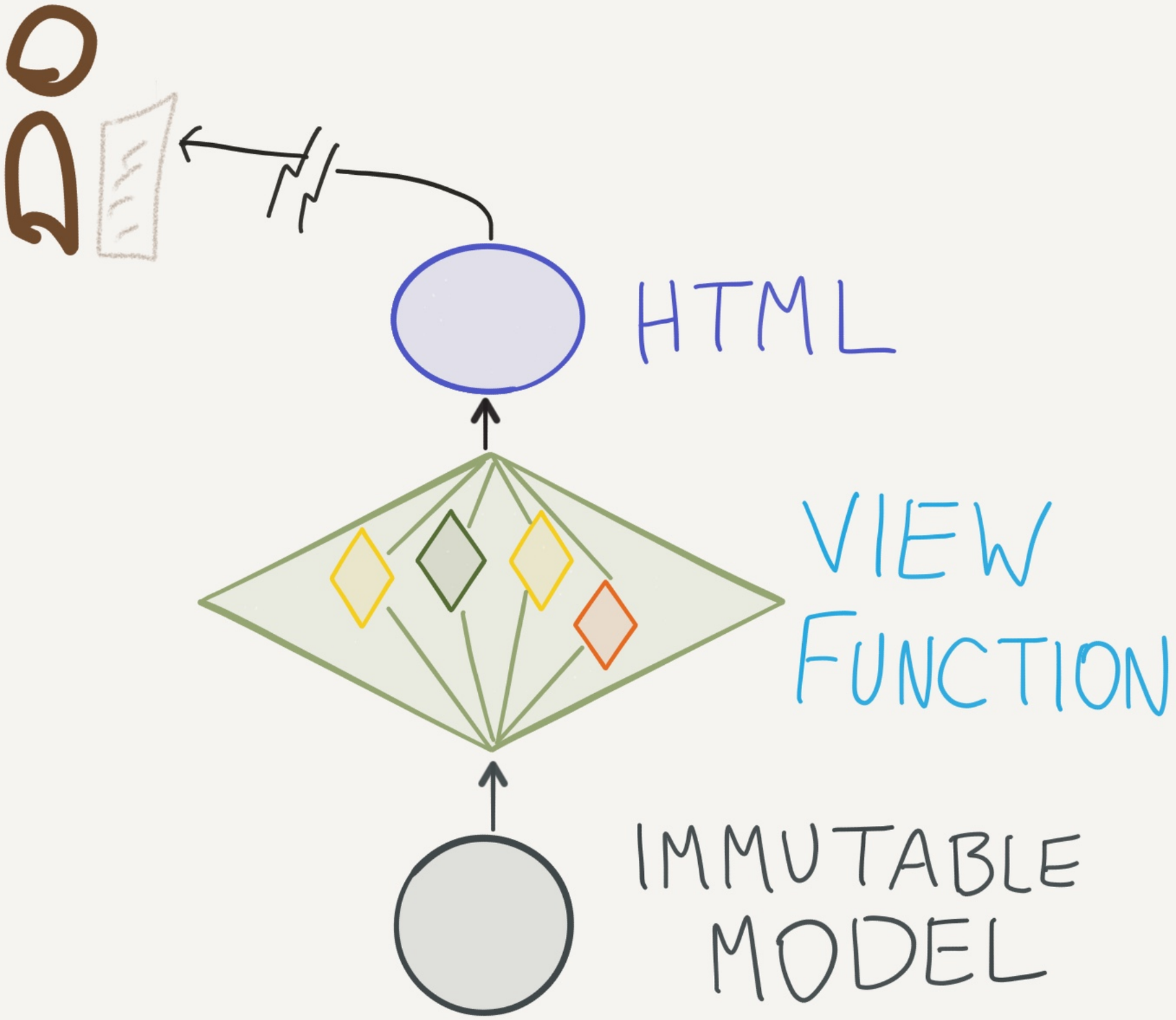
A precise estimate is BS, but we can say "at least 2 weeks, and I would be

embarrassed if it took more than 2 months."

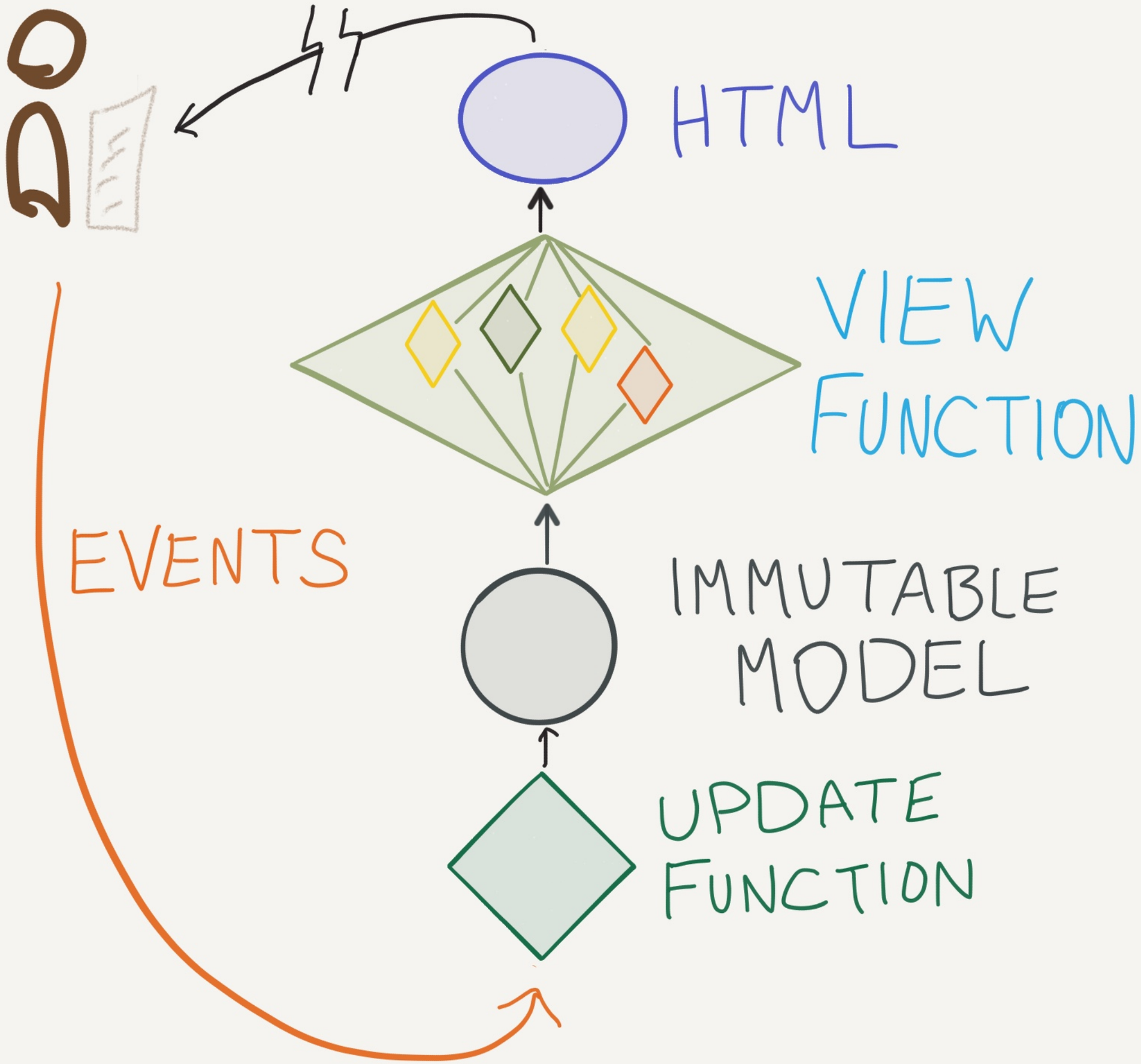
Then in early work, we can look at the riskiest parts of the project first, learn more about them, and narrow the range.

What Comes After MVC?

Answer: the Elm Architecture.



Elm is a front-end functional language, compiles to JavaScript. An Elm application has one model, and that model is shared (or parceled out to) all the views.



The views are open to interaction with a human. The person clicks on something, an event is generated -- and it does not go right back into the view. Instead it comes around to the one function responsible for

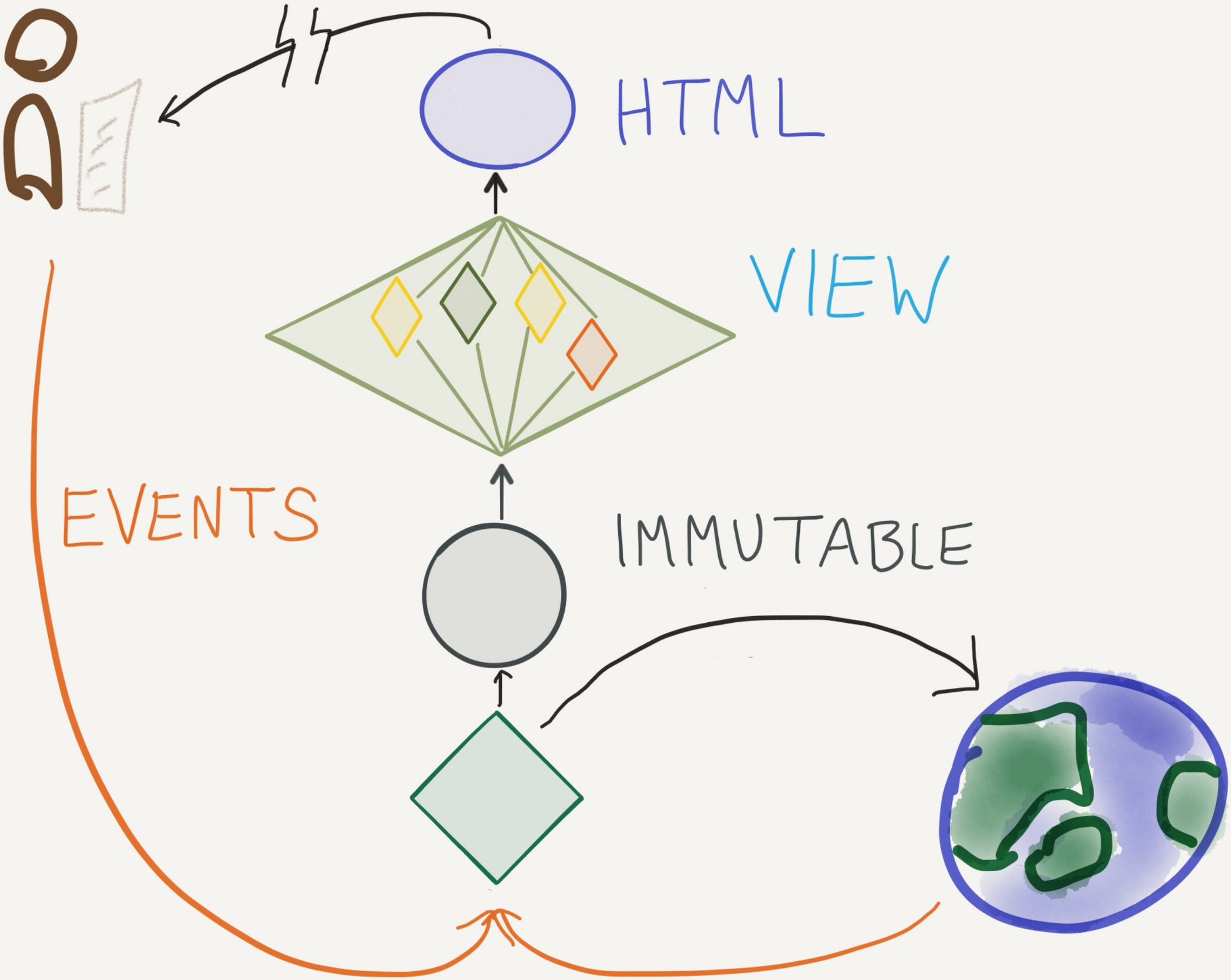
decisions that update the model. The update function receives the prior model contents, plus one new event. The update function decides what the model should be now, and a new immutable model comes out.

The view functions, with the new model, provide new HTML, which is diffed against the views before, the DOM gets updated, the next event is applied. There is a loop. Exactly one. It is traceable, you can record the stream of events, and the model values that result.

This is an idea coming from many sources. React, especially with the Redux implementation of Flux, has this structure.

CQRS says, "writes and reads are separate." Say no to two-way bindings!

Event Sourcing is all over this. Present is a left fold of past state.



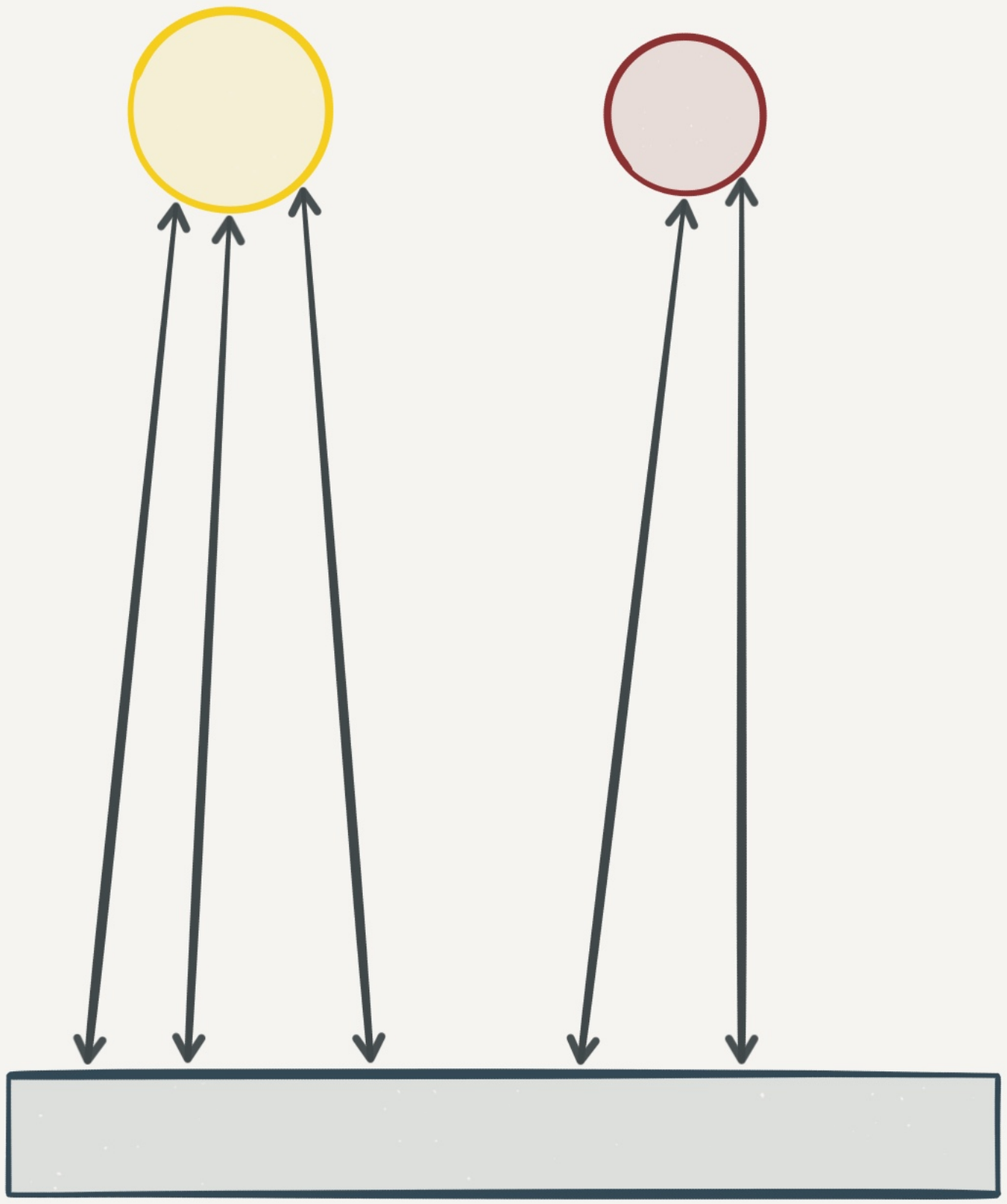
To access the server, this update function can also return instructions. Elm has a strict separation between stateless, functional code, and JavaScript libraries and interop, which can access the outside world.

Anything the server sends back comes in as an event, and goes to the single point of entry, the update function. It has a very clear structure.

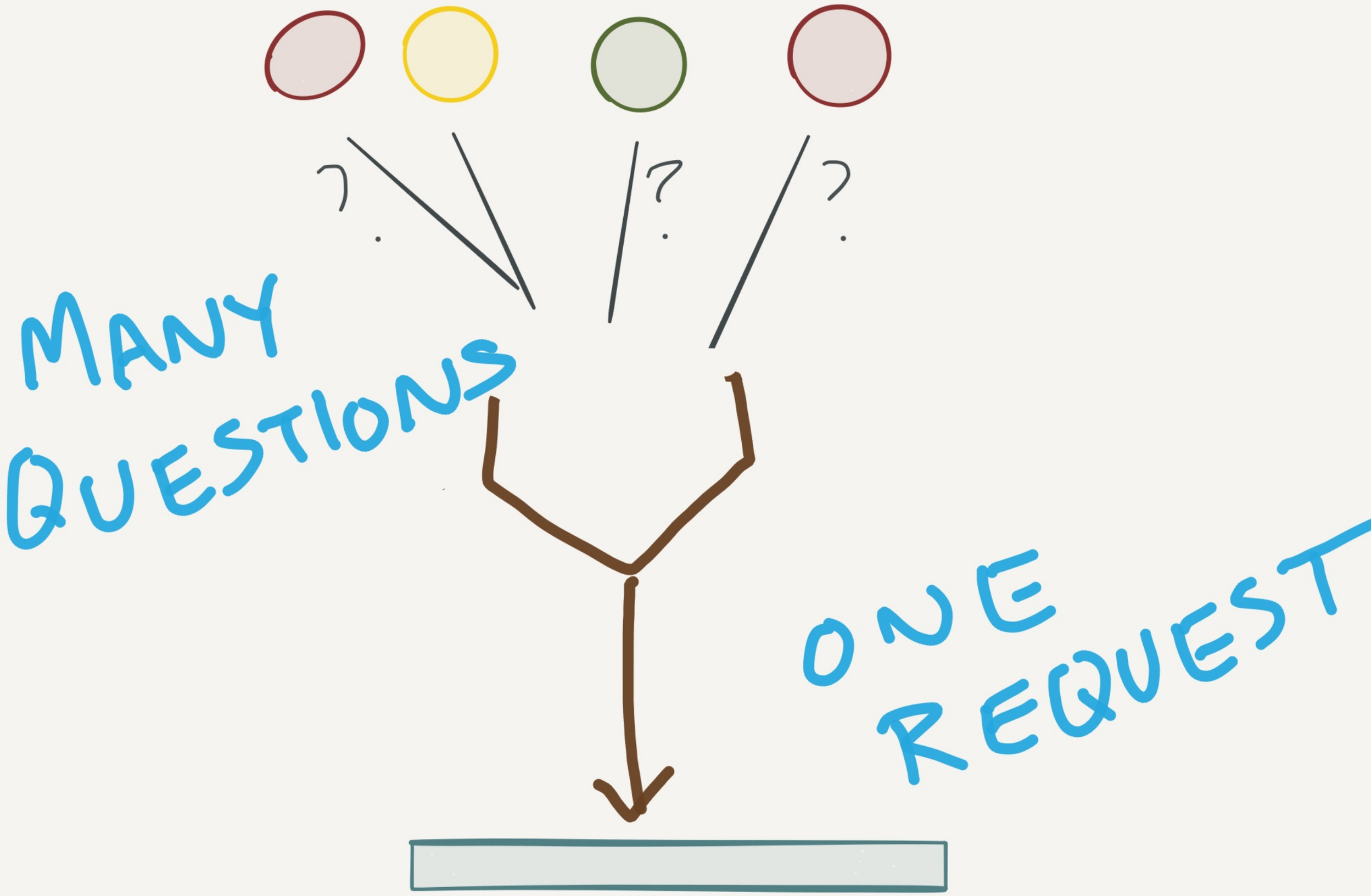
One thing that gets annoying is making calls to the server based on responses from the server.

What Comes After REST?

Answer: GraphQL.

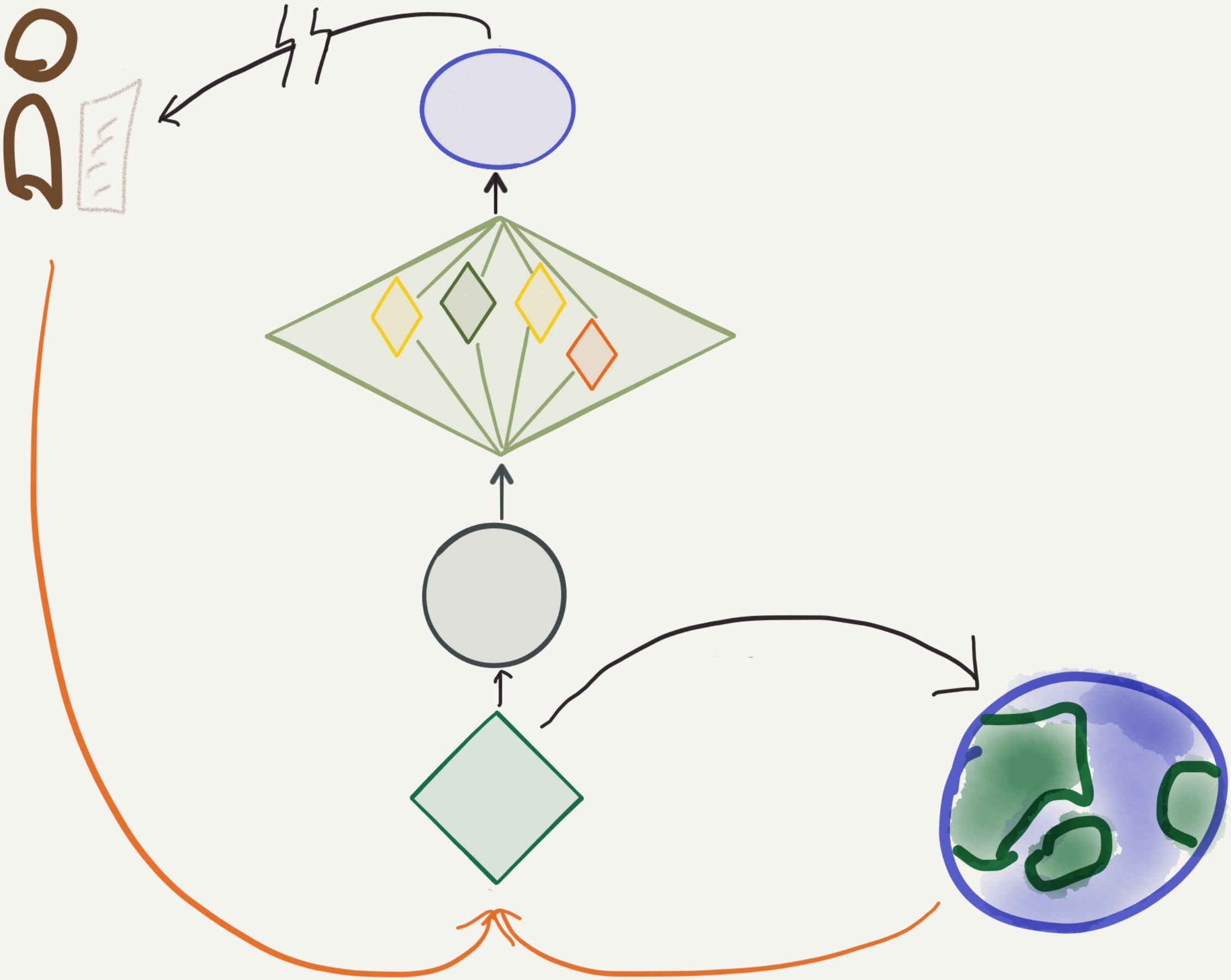


REST says, start from the top level. Make calls for each level in the data graph as you go down. Ask for the conference info. Then ask for attendee info. Then ask for details on each attendee you display.

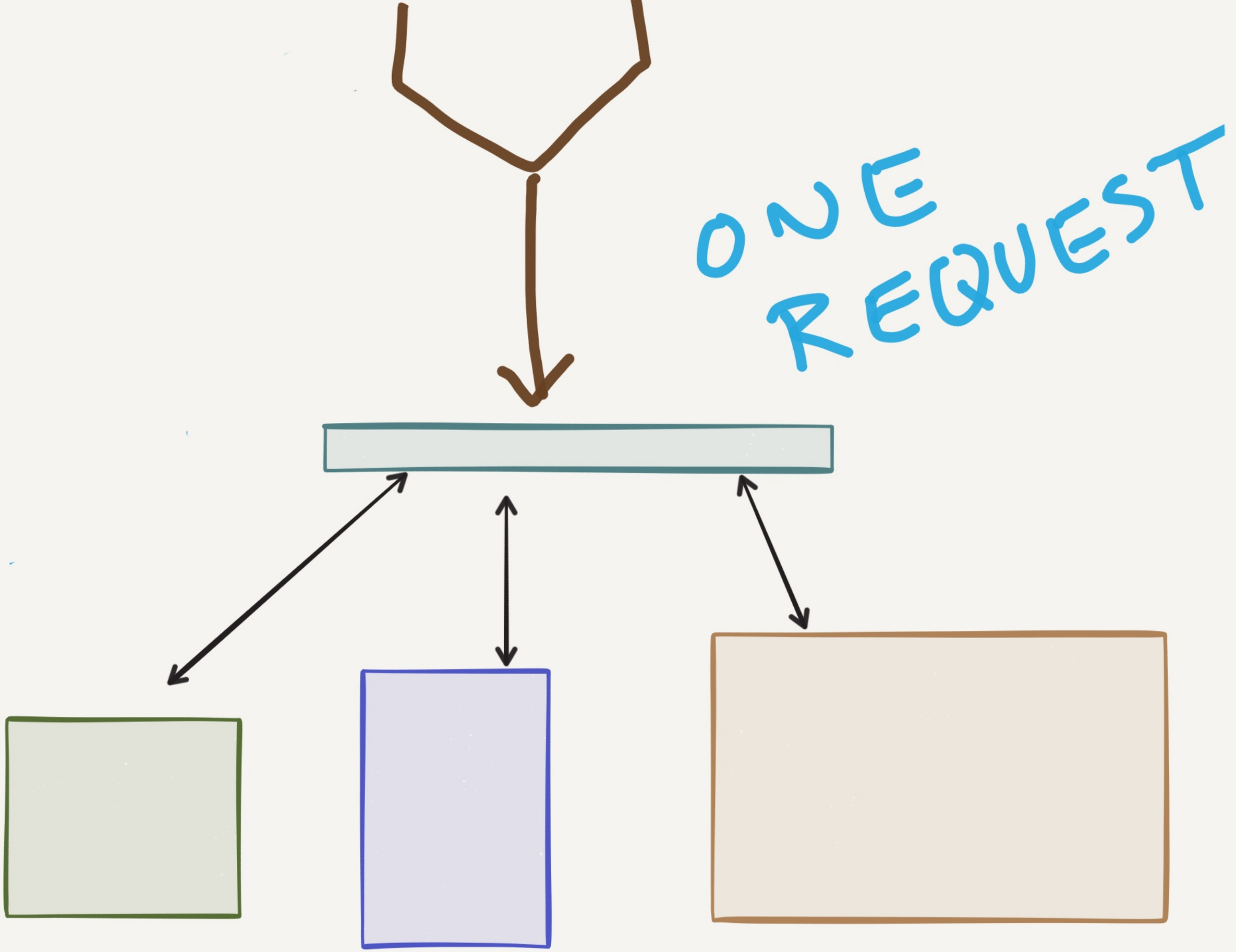


GraphQL says, each view component, declare what you want to know. On the client these compose to one request. That request goes to the server, and the server responds with the requested data in the requested

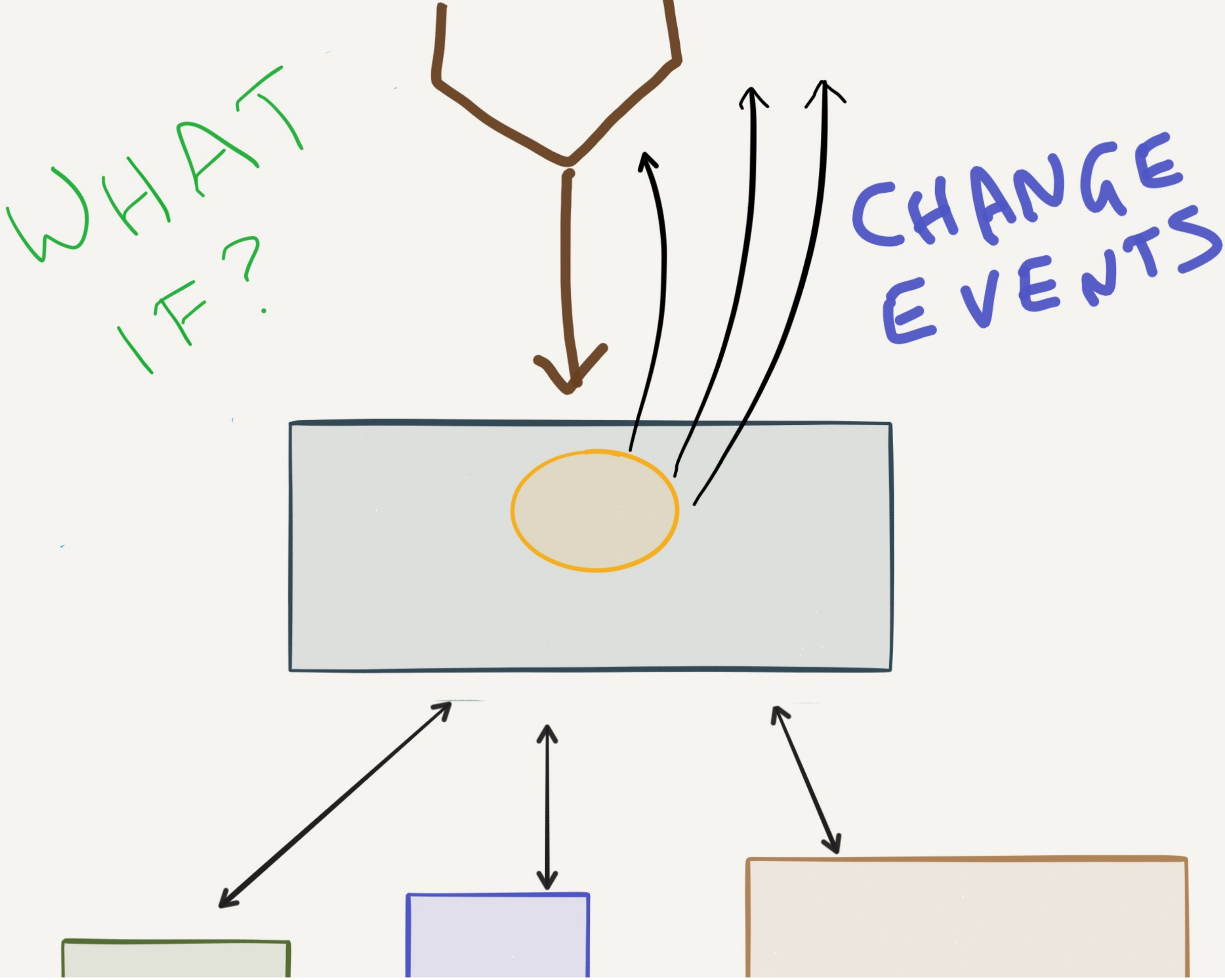
structure, and no extra data.



This fits beautifully into the Elm Architecture, because that one response from the server fits into the model that determines all the views.



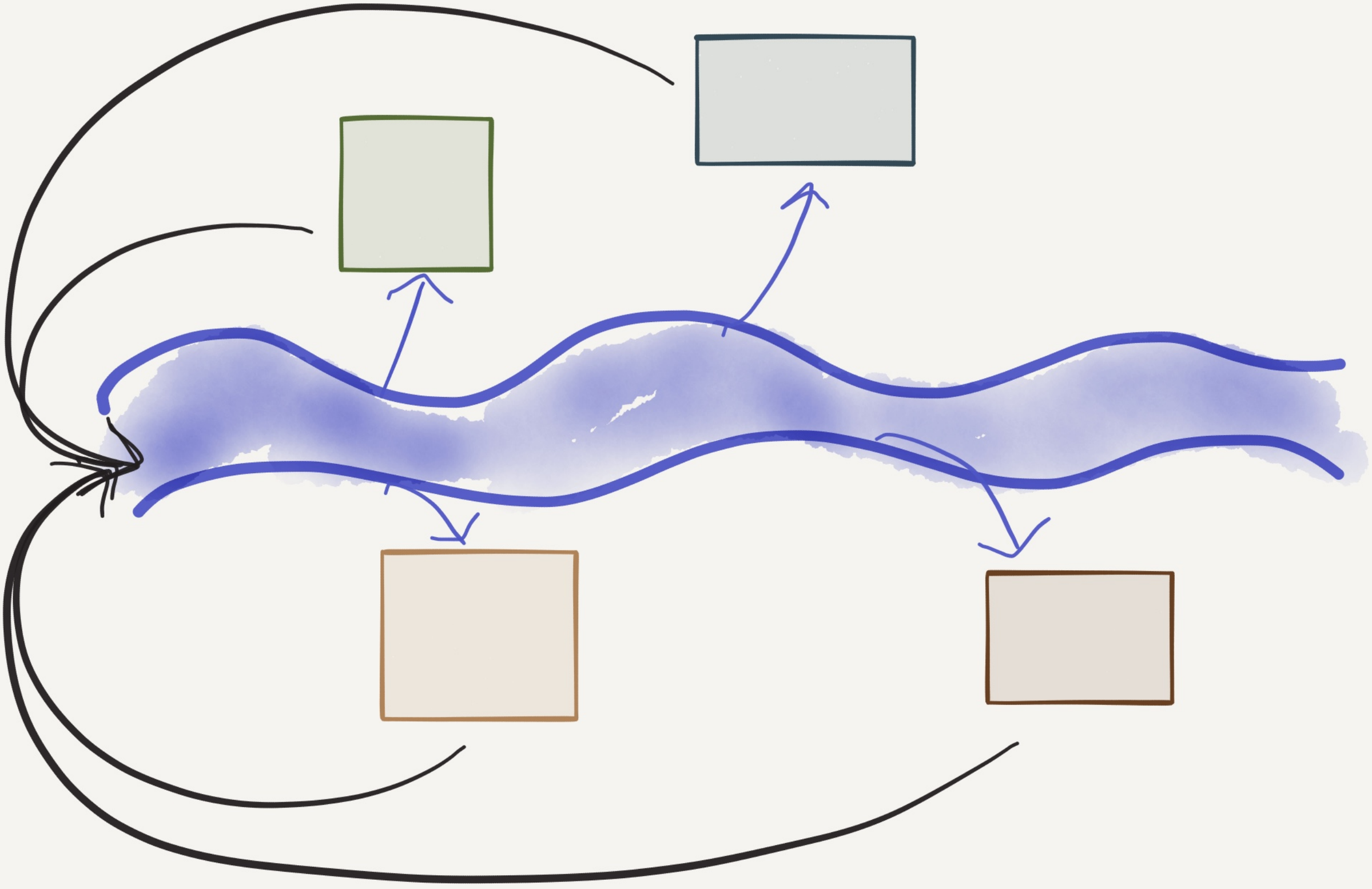
On the back end, that GraphQL server can farm the different parts of the query out to various services. It assembles the results for the client. One response.



This isn't part of GraphQL yet - but what if that one request opened a web socket, and the results come back as they can... And it works as a subscription to any updates to that data?

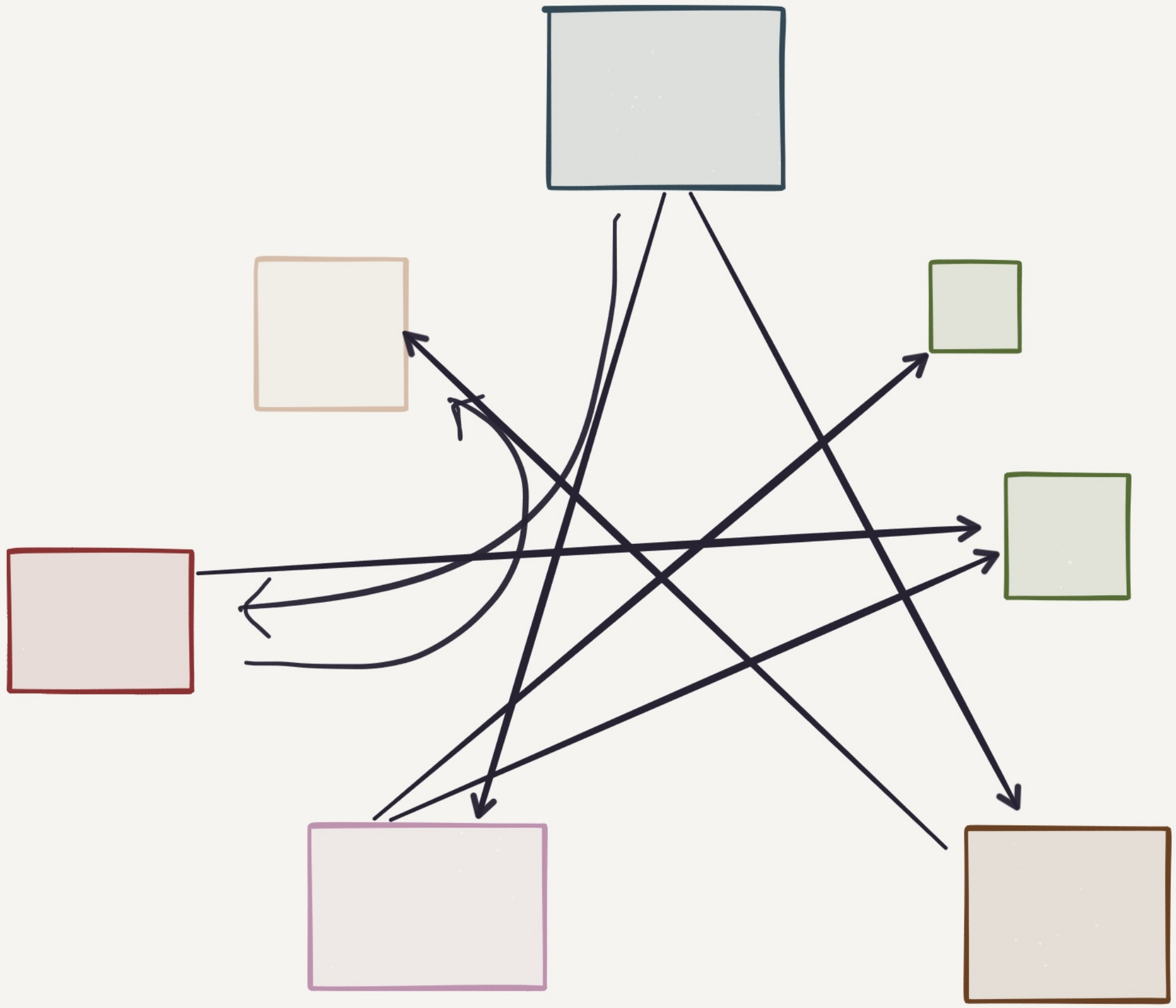
WHAT COMES AFTER MICROSERVICES?

Answer: groups of microservices. In Elixir??

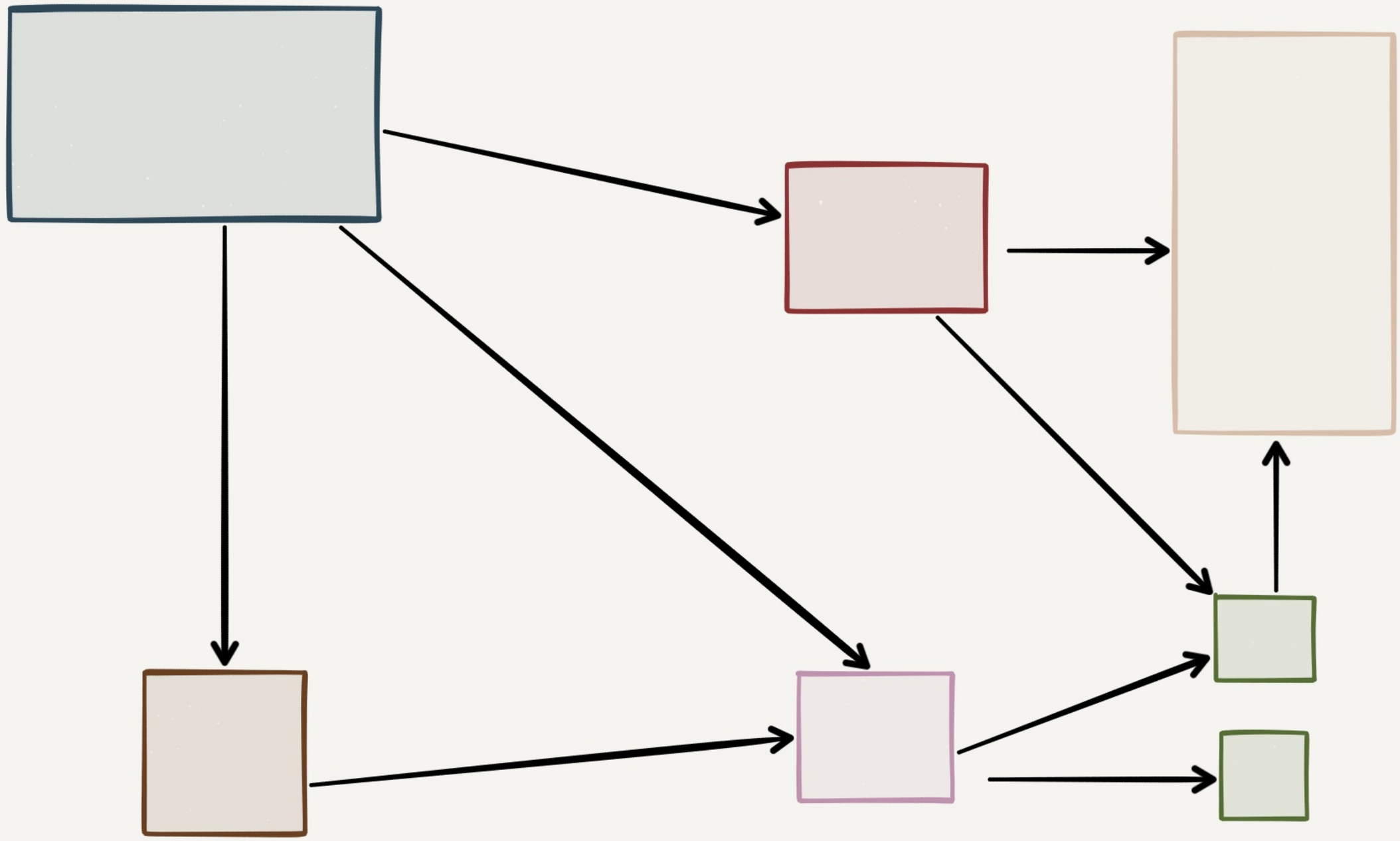


This is the original microservices architecture espoused by Fred George, back when no one had heard of microservices.

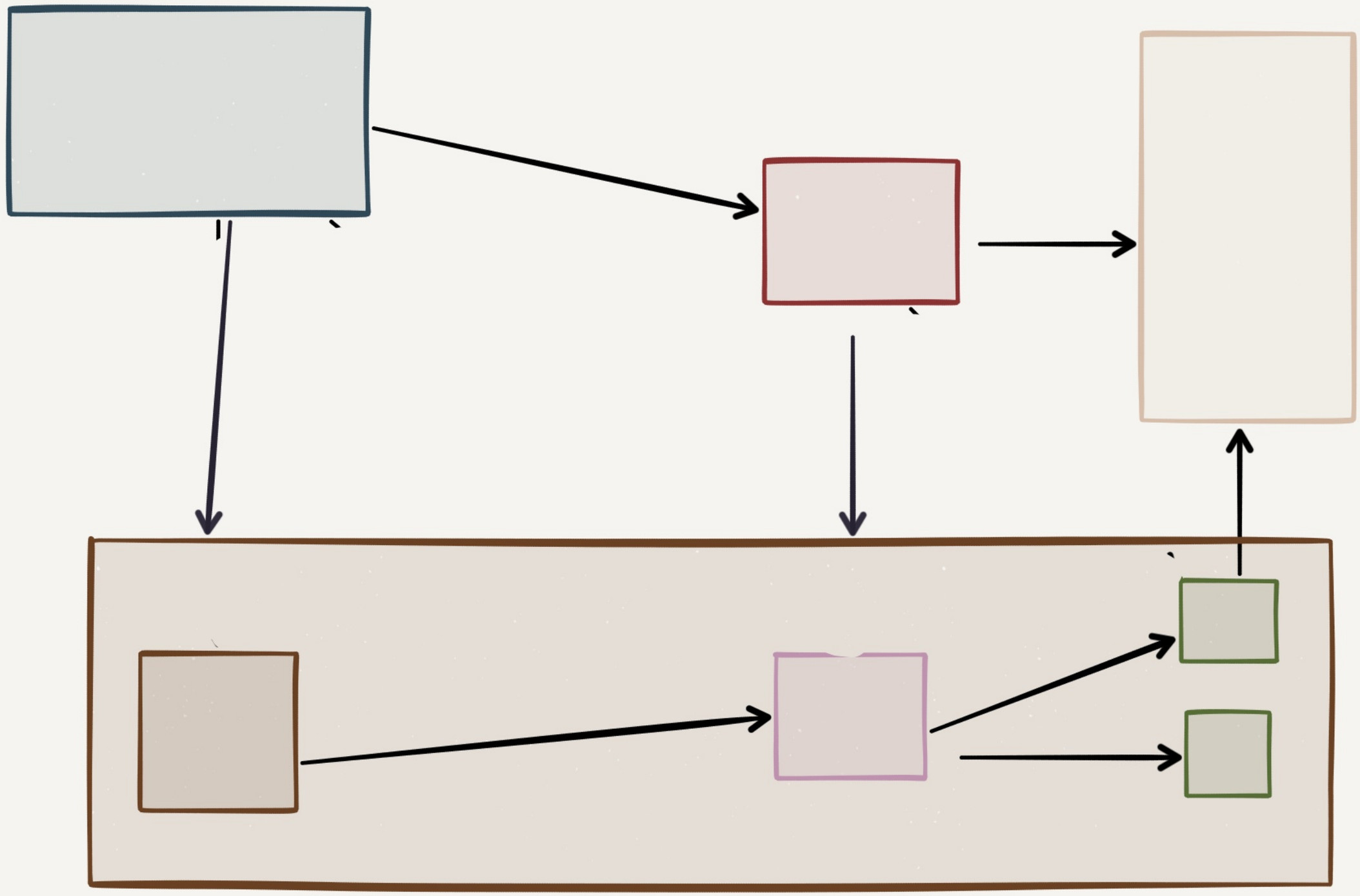
There is one giant stream of messages. Microservices read a filtered stream of messages, and dumped their responses back into the river. It has that single point of incoming change. It also doesn't scale well.



So we let our microservices talk to each other directly, and we get this.

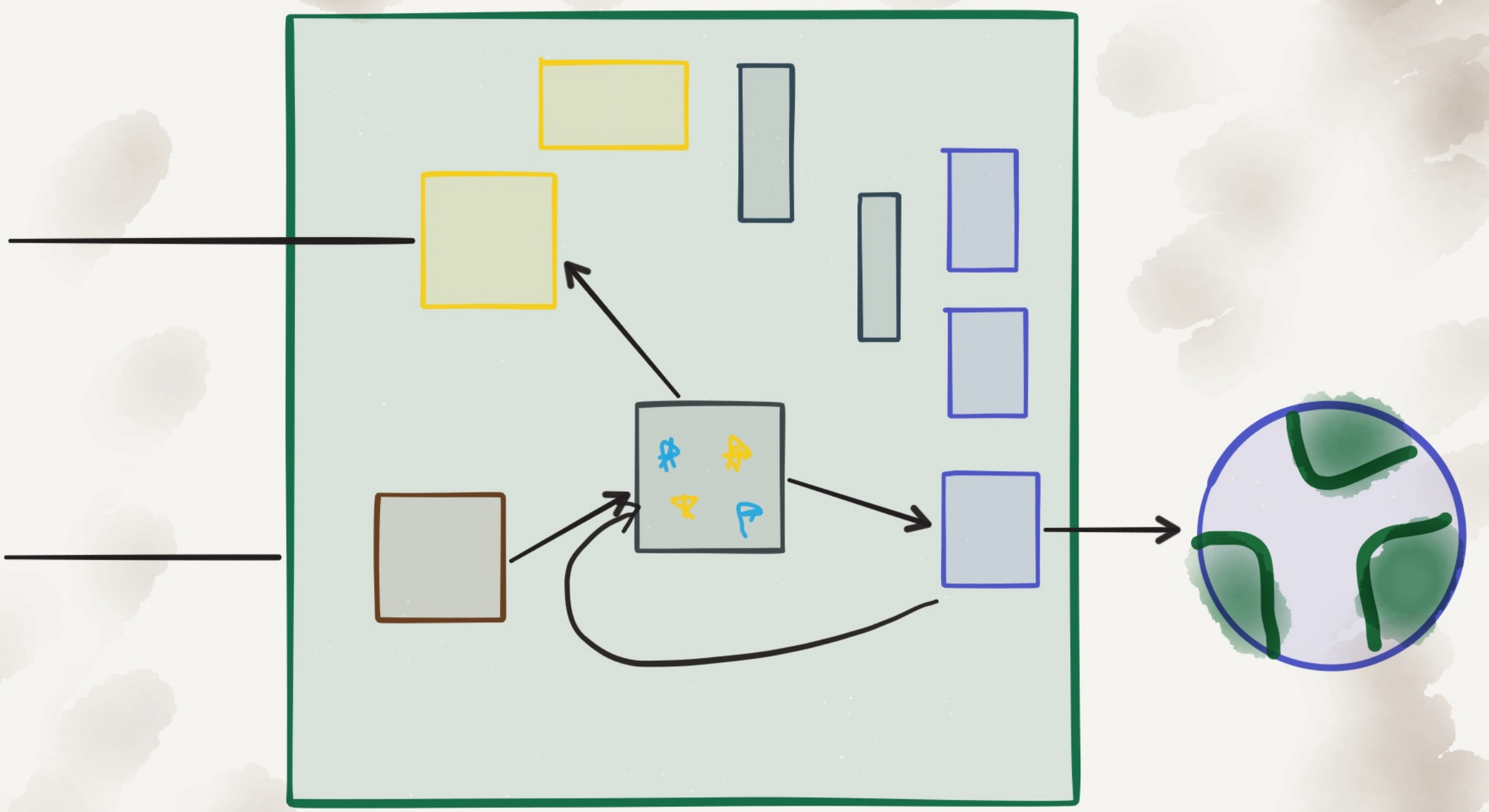


But if we look for a directed acyclic graph, and aim for it, we can find it in that diagram. Untangled.



Then we can group services together, into one larger application.

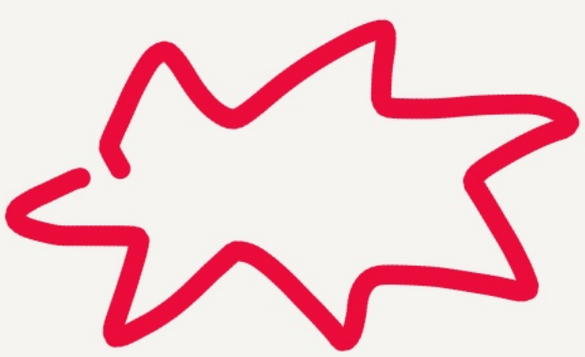
Ecosystem. Constellation.



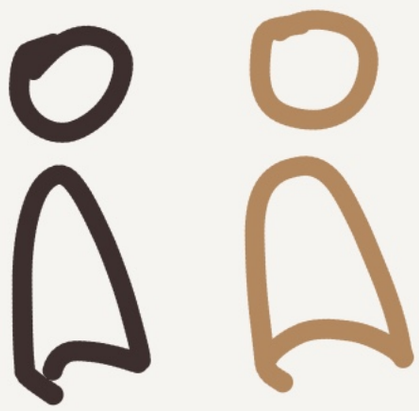
Also, within one service we can break it down. Something I love about the actor model: an actor can really be an actor subsystem, and that's invisible from outside. The structure can go as deep as we need it to.

FP: immutability

OO: independence



: isolation



: people!

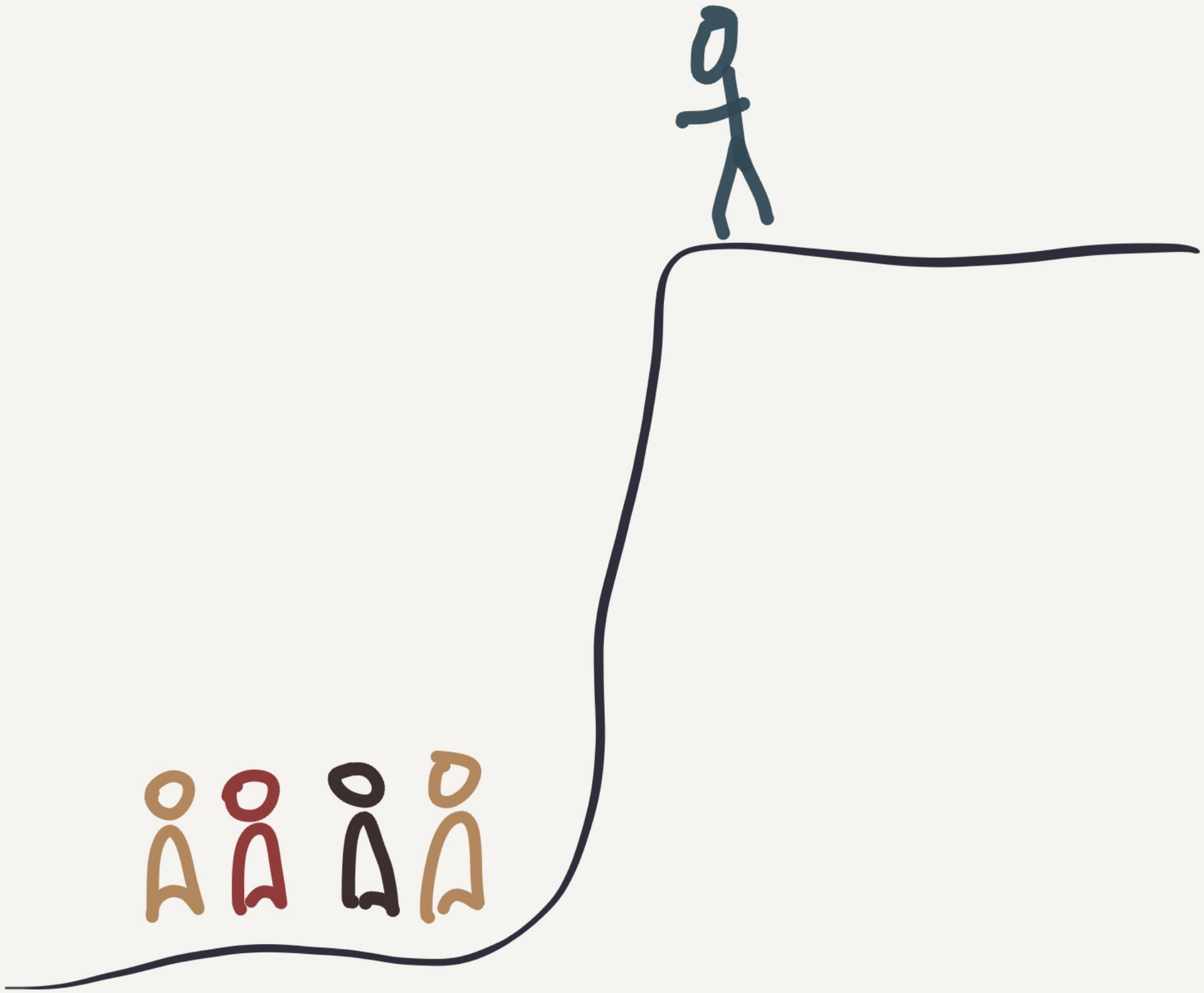
Elixir takes some of the best from FP. Immutable data. Flowing data through transformations, like web requests to responses in Phoenix.

Elixir takes the best of OO, in its original intention. Asynchronous message passing means our processes are independent.

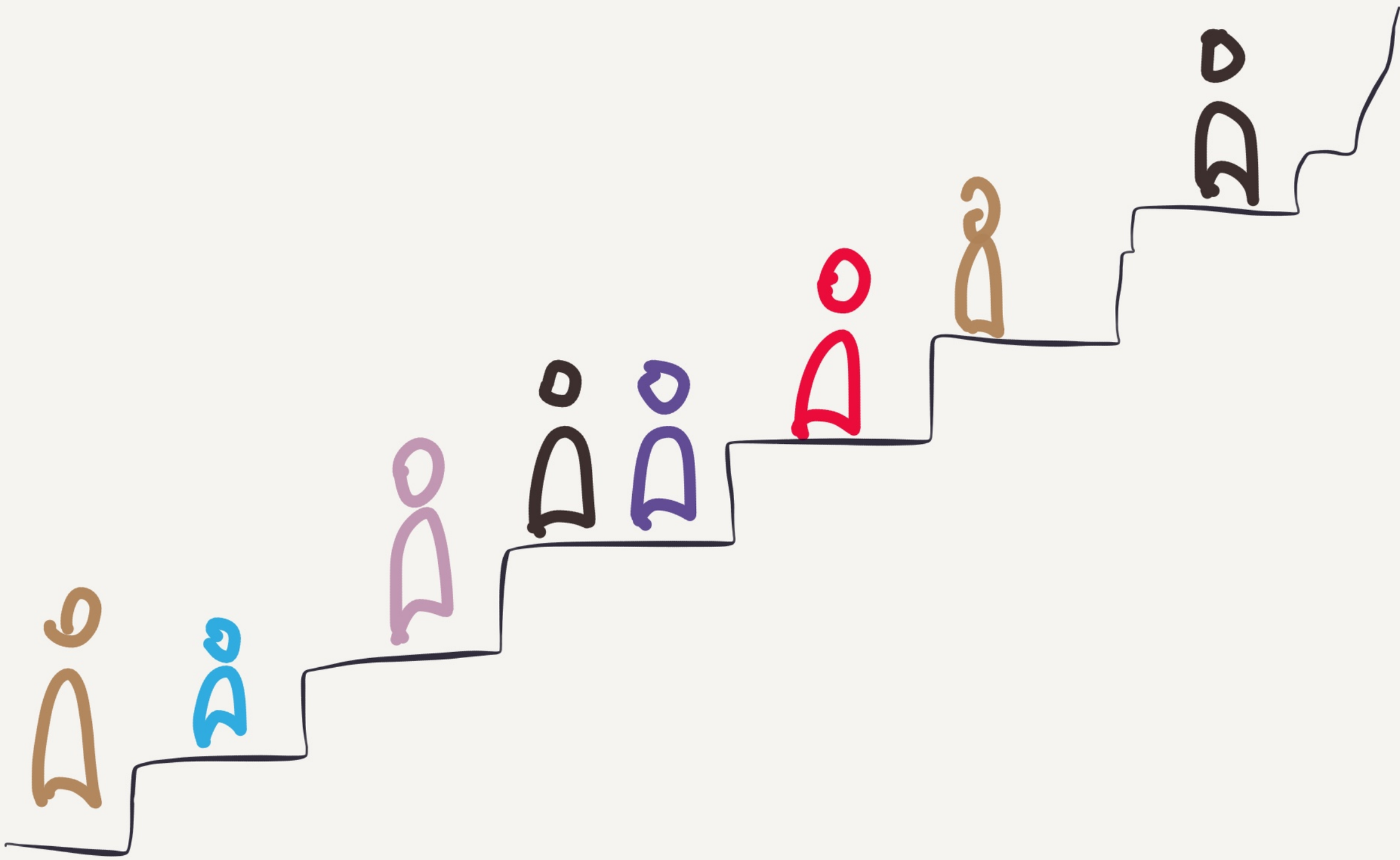
If we are careful, we can avoid circular dependencies, making our code more functional-y, and avoiding the chaos of many microservices implementations. We want lots of feedback loops in our teams, few in our code.

Elixir handles failure better than most microservices implementations, too.

All these things are cool, and they give Elixir great potential as a programming system. But what determines whether it will reach that potential? What really matters, what makes a paradigm and makes progress, is people. It's us.



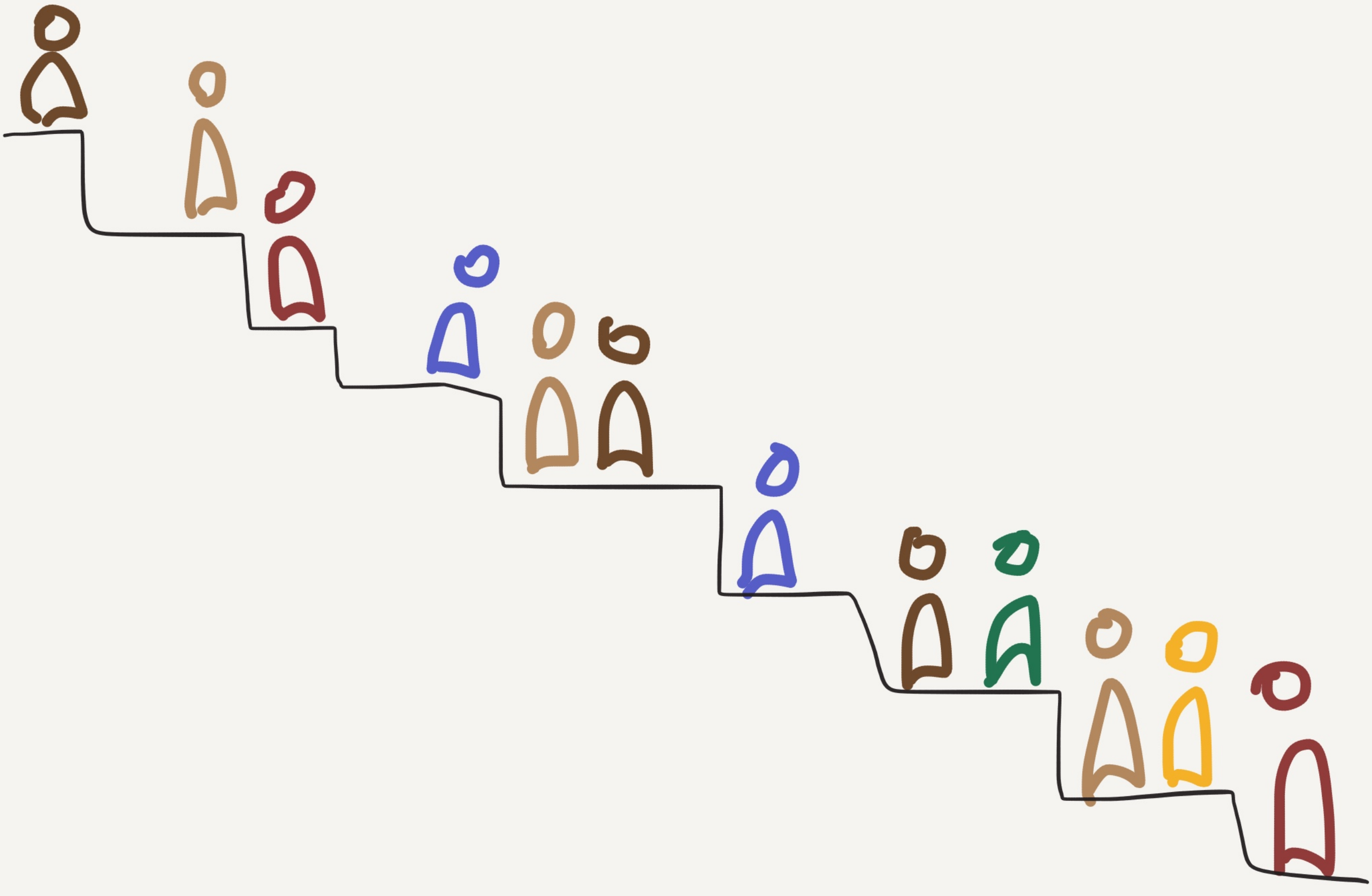
Some programming languages are unapproachable. The experts are experts, and if you want to be accepted into their ranks, you're gonna go through some tough times.



Other languages have people, documentation, communities at every level. Ruby is like this. There are beginner materials, user groups, friendly conferences, and people at many levels of expertise who help

bring others up. Especially at the beginner level.

Elixir can have this. Josè sets an example, he and others have created a friendly language, with lots of tutorials and docs. We need to continue this. In particular, as the ideas keep coming in, as Elixir sphere gets stronger and stronger, keep these stair steps short.

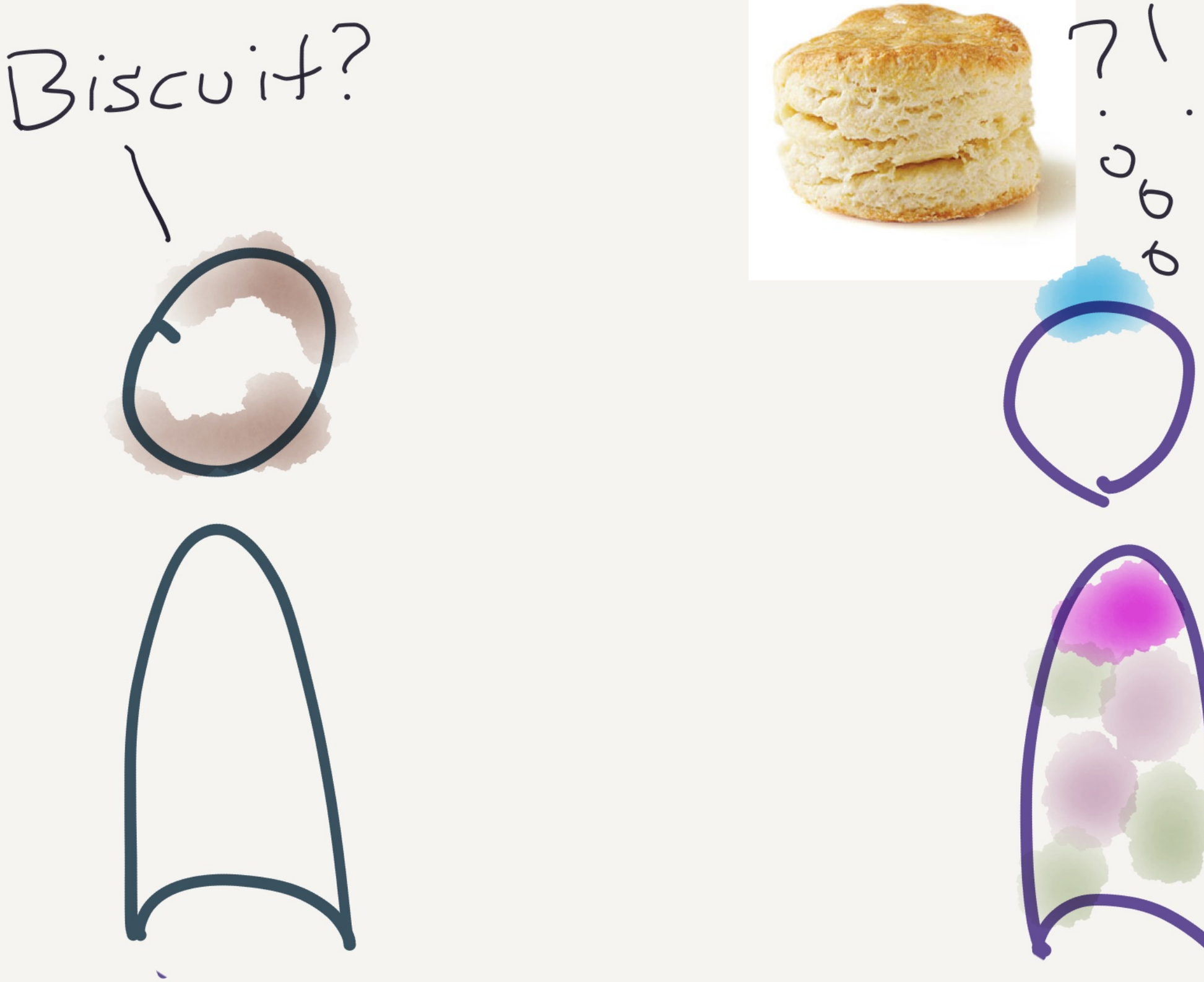


And it isn't only one path! There are people from Rails, people coming from Java, people who have no programming experience. Each of these paths is different. If you are from one of the less populated paths, your

explanations and contributions are even more valuable.

Because between paradigms, it's hard to communicate. The same words mean different things. What seems obvious to you is strange to people who don't share your experience. Plug is clear to functional programmers. MVC and templates make sense to Rails devs, not to me.

This is a problem in science too. Gravity doesn't mean the same thing to Newton and Eistein models. Property doesn't mean the same thing to a functional dev vs OO. Nor "persistent data."



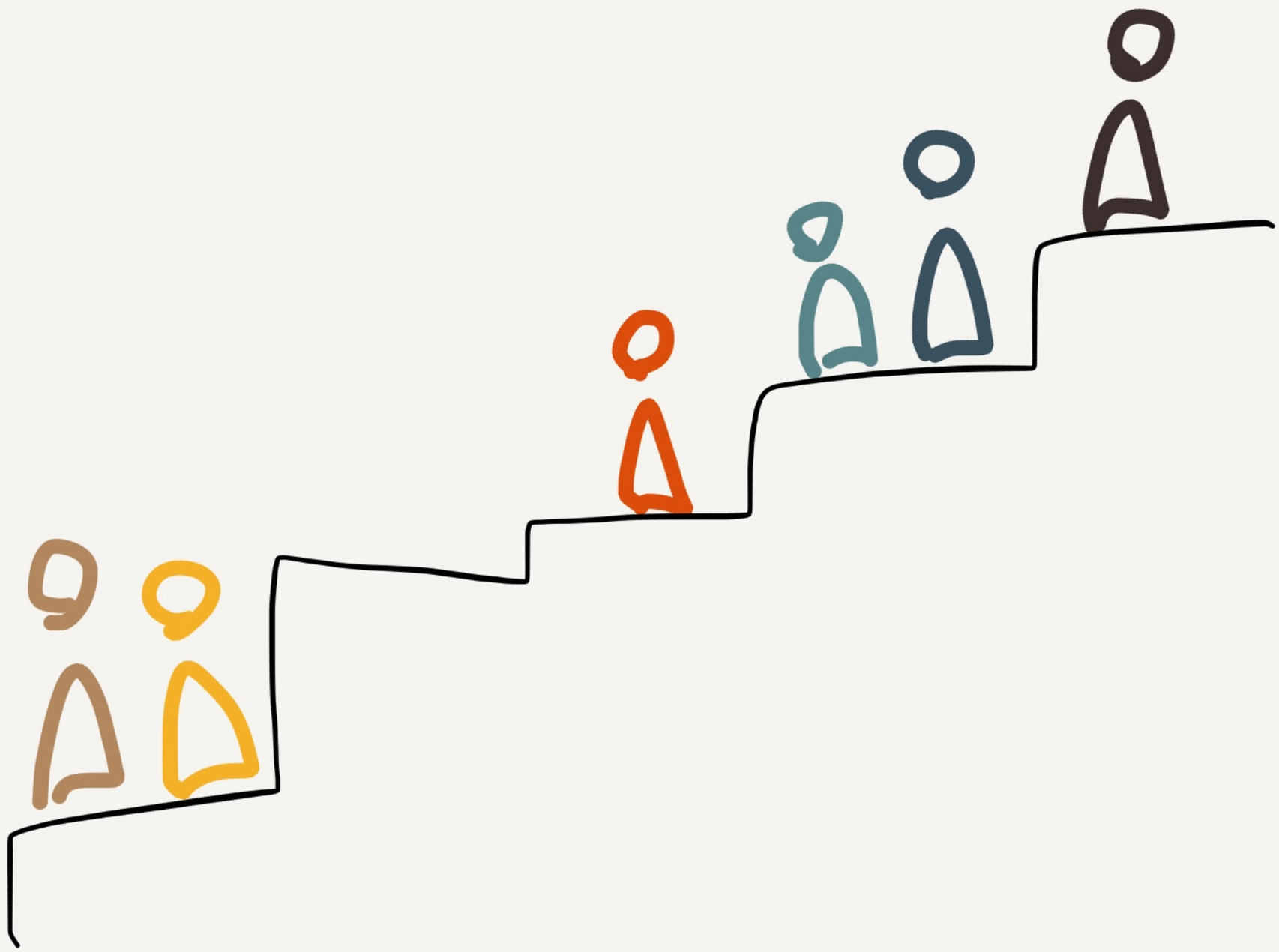
Alan says to me this afternoon, "biscuit?" And I was confused

Until I remembered that in the UK a biscuit the name for what we call a

cookie.

This is a geographical difference in the language, but the same problem exists between people who think in different paradigms.

Newton and Einstein didn't mean the same thing when they said "gravity" or "mass," even when they used the same thing in equations. A paradigm comes with its own vocabulary -- functional programmers talk about "currying" and "fold" -- but it's worse when we use the same words for different things. Like "property" and "persistent data" and "map" - these mean totally different things between FP and OO. Not to mention "process" and "application." This is another reason it's important to have a variety of learning experiences. You have a set of experiences that is going to help other people if you share it.

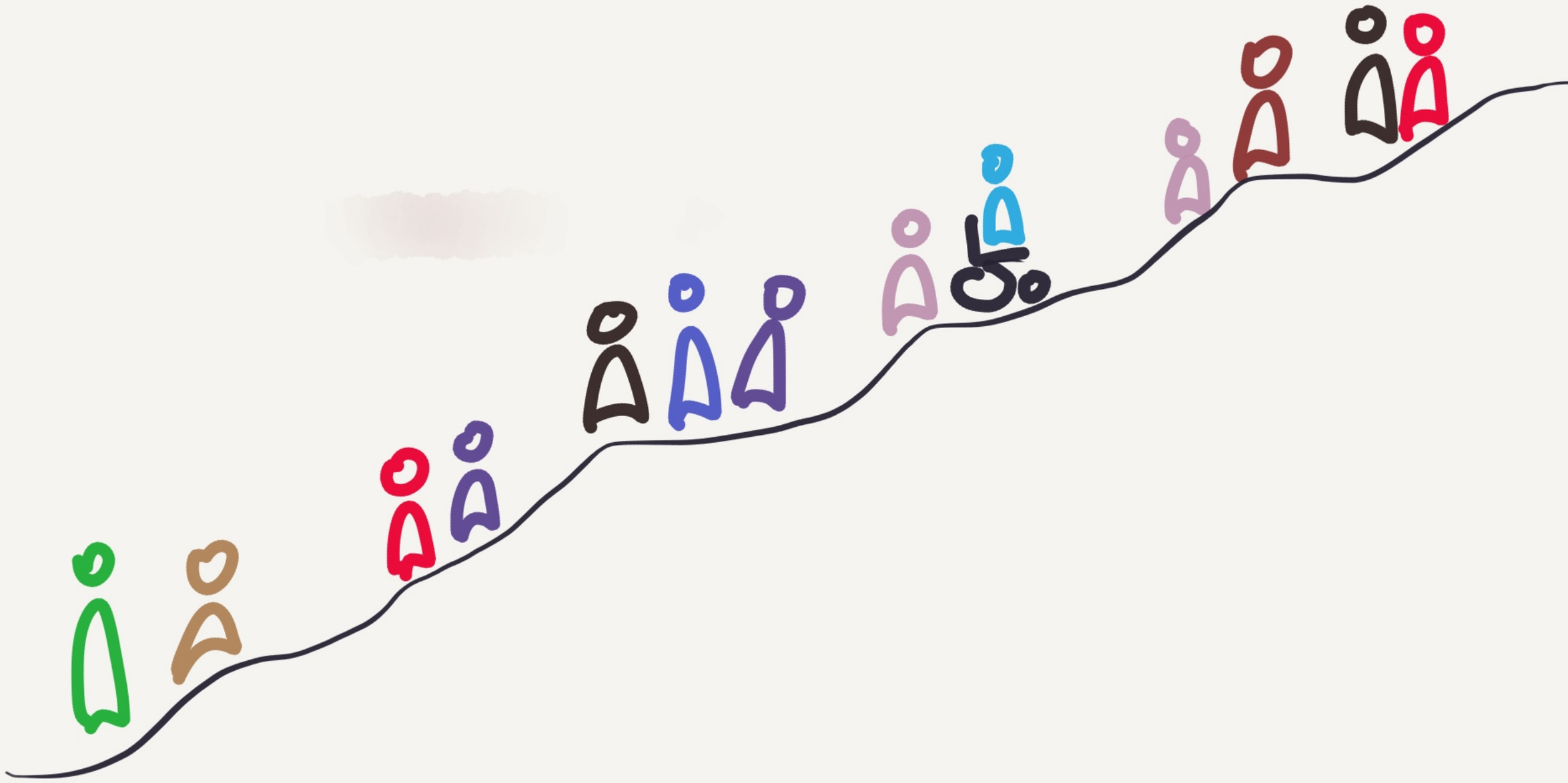


And people who aren't yet programming. They have a longer road, and yet they're the most valuable. The new ideas, the big paradigm advancements, usually come from someone outside the field. Kuhn was

not a historian. Franklin was a politician and many other things. More new ideas come from here.

Also, as professional developers. I fell into programming and it wasn't particularly hard. I don't realize how much I know. We complain about recruiter spam - that's ridiculous! Oh poor babies, we have too many job opportunities! Easy for us to talk. People are trying now to break into this field. We are standing on a glass floor. We can choose to stand here and feel smart, for being lucky. Or we can help. We can leave breadcrumbs - blog when we get an error. Or ask a question on stackoverflow and then answer it. We can answer questions. Most importantly, we can be nice, and make people feel like they belong, like they're welcome. Because they'll bring their contributions, and their ideas, and Elixir will be that programming system that lets people build beautiful interactive apps without the resources of Google. And it will be the starting point for the next big ideas.

Encourage each other. Ask questions. Set an example: say "I don't know" whenever you have the opportunity. There is no guilt in not knowing. Asking and answering helps everyone.



No matter where you are in your learning, blog it. People are coming behind you. If you make it just a tiny bit easier for all of them, then Elixir will grow. New people and new ideas will join us. Hobbyists will become

professionals. Publish your code, read others' code, discuss which style is clearest. This lively community discussion is what makes a living paradigm.

An idea like this doesn't belong to the person who thinks of it. Nor the first person to publish it. To bring an idea alive, to make it useful in the world, it takes all of us. All of us being a little irrational, taking the time to

blog an error after we fix it. Moving in the interest of progress rather than personal efficiency.

If Elixir is going to take over the world - and push programming toward the next big paradigm of usefulness - then we are going to take over the world. Not a hostile takeover. We can win hearts by welcoming everyone.

Great code doesn't come from a place of derision. It comes from inclusion and welcoming. It comes from people who are crazy enough to come together and share, with each other and the world. Who are

bananas enough to run a conference, or answer questions on Slack. To report bugs and submit pull requests. To read lots of code, and review pull requests. To create editor plugins and keep them up to date. To write, write, write and publish. Keep participating. Because elixir and mix and hex are your ideas too, if you build on them and bring them to the world.