Jane Street
Tech Talks

How Jane Street Does Code Review

How Jane Street Does

Code review is a fundamental part of developing high quality software, but it's also a difficult practice to apply rigorously.

As a team grows larger and its codebase grows more complex, it becomes harder to maintain the appropriate level of discipline: merge conflicts happen more frequently, branches need to be rebased more often, and sometimes the most delicate code changes receive the least amount of scrutiny.

This talk will describe how Jane Street solves the problem of scaling code review to a large team, while ensuring that all changes to production-critical code receive careful attention. We'll focus on Iron, the code review and release management tool that we've been using and improving internally for years.

We'll show how Iron models the process of code review, and how this model is sufficiently expressive to handle messy cases like merge conflict resolutions and reviewing code across rebases. We'll also talk about some of Iron's other features that it easier to develop robust software, like its hierarchical feature branches and per-file scrutiny controls.

TRANSCRIPT

So, we're going to talk about how Jane Street does code review. Specifically mostly in the how department. I'm not going to motivate code review too strongly. I think that if you're at a talk called, How Jane Street Does Code Review, you probably appreciate the value of code review a little bit. So I'm going to go right into that and talk about the tool that we wrote. But not as much the tool, so much as the ideas that underline our design of it. Because you're probably not going to use our code review tool. It's a weird in-house thing, but you might hear some of these ideas and think, "Oh yeah, it is kind of weird that code review tools don't do that, and maybe

it, but I was not involved in the design. No one who was involved in the design is here in fact, so we'll have an interesting Q&A session afterwards. But, Iron is this in-house code review tool that we wrote. And when I started at Jane Street, that phrase was alarming to me. When I picture an in-house code review tool, I think of this very ugly web tool with programmer CSS that kind of works sometimes. And it wants to be GitHub/GitLab but it's just a worse version of all of that.

And it's actually a completely different thing. It started from different first principles, and it built out a separate tool. As I've come to use it, since I started at Jane Street, I really like it, to the extent that I'm giving a talk about it now.

Iron is open source, but like all Jane Street open source projects, it's a weird, dark kind of open source. We publish all of our code to GitHub but we develop all of our code in Iron, which is a Mercurial-based weird other revision control system. So if you actually go and look at it, you'll see this hostile series of commit messages that are version numbers and these massive diffs. And if you actually open and look at the diffs, you'll see that it's all OCaml code and it's very hard to actually open the source code.

And there's no install instructions. There's pretty much no chance that you're going to use Iron, although if you're interested in actually running it, we should talk about that, because it would be an interesting puzzle to learn how to set up an Iron installation. We're going to focus much more on the ideas. This isn't a sales pitch for the software. You'll never use it. You're probably not going to use it, unless you're going to work at Jane Street.

And I hope that some of the ideas maybe strike you, and you want to go and build tools so that if I ever leave Jane Street, I don't have to give up all of these things that I have … Yeah?

Do you have documentation for this inside the GitHub?

you look at files, maybe you look at individual commits. There are different details but the core idea is the same. And if any of the ideas that I'm presenting, if you're like, "There actually is a tool that has this idea," then cool. We should talk about it later.

I was very impressed that Jane Street's tool gets away from this. A lot of the ideas I'm going to present to you are worse. This really is a local maximum situation, where things are going to get a lot worse before they get better. And only when you see a lot of the ideas together do they really make sense. This is going to be a theme that comes up as we talk about different features.

That's a really boring graph. So, just as a motif for this idea, you're going to see a local maximum lizard throughout the talk, because you need something to make it more visually interesting.

Okay, so part one. Actually reading code. This is what review is, is you need to read code. And this is really difficult. Doing a good job is especially difficult. Actually analyzing code, you can look over and make sure that it's following your style guide or whatever. But that's not really valuable. The more work that you put into it, the more you get out of it. You can really deeply understand the code, the better you understand the code that you're reading, you can offer more suggestions, improvements, maybe ideas. There's this big tradeoff towards understanding code and getting something out of code review. You get out of it what you put into it.

I think code review is one of the harder things that I do as a developer. And you might imagine that we're making some tools to make that easier. You can imagine a semantic diffing tool where you can view your program at a higher level. And instead of diffing files, you can look at functions and modules and you can imagine making code review easier in that way. And we're not going to do that at all. We're going to focus on a much dumber thing, which is mechanically reading code and understanding it, is a difficult thing to do.

I'm going to talk, just go through a simple example. It's illegal to give

into your branch, then if you were reviewing that branch or in the middle of reviewing that, this is this weird situation, because now you have a merge commit. Do you review the merge commit? Well, the contents of it are, it contains everything that happened upstream and not really anything to do with your branch. But maybe. It contains real semantic changes because it's a merge commit. You can do whatever you want.

This is something that, it would be nice if you had a review tool that could handle this case. Merge in trunk, symmetric operation. Another thing you might see is, you have a branch that falls behind its master, and you want to rebase it. So, essentially you create a clone of it. You replay all of the commits on the new head. The old commits are still there, but they generally get garbage collected. And then you can release the branch in a single step without the merge.

This is a very personal question, whether you merge or whether you rebase. It would be nice if we had a tool that could work with both of them. They both have their own difficult problems. When you're rebasing, you have blown away all of the commits that you had, and all the commits that you have reviewed, and you've created a brand new set of commits. And maybe they're the same, and there was no problem, but they could be arbitrarily different. And you can do whatever you want in the course of the rebase, which is kind of scary. If you're reviewing it and it gets rebased, you can re-review the whole thing, but that's unattractive. You can trust that the people who rebased had done the right thing, but none of this is very nice.

Just as a quick aside, at Jane Street we always merge, but we call it rebasing. It's a nice compromise between the two, which goes into something that we'll talk about a little more, which is that we think a lot more in branches than we do in individual commits. The argument of rebasing as a nice clean way to preserve a clean history, it's something that we zero percent actually do at Jane Street. We

then you have a whole new set of commits. It's difficult. And I've never used a tool that really solved this problem.

The general solution to this is, don't rebase while you're doing a code review, that I've encountered. Don't review merge commits. It's probably fine. Trust that everything is okay. This is what most tools that I've used have adopted. I would be interested if anyone else has seen a decent solution to either of these problems.

The solution two is, we could actually come up with a list of constraints and then find some way that we could review all of these things. And that's roughly what Iron does. I'm going to come up with an artificial list of constraints that I want my code review tool to satisfy, and then magically, I think that it's going to satisfy them after I'm done.

We want to review all of the code, including merge conflicts, including if we rebase and we ended up changing something in the middle of a rebase, we don't want that to go by the wayside. So, a very easy thing to do is just review the entire state of the entire code base anytime anything changes. But that's terrible, so we don't ever want to duplicate review. If you've already reviewed a change, we don't want to have to review it again.

We would also like to be able, while we're in the middle of reviewing a branch, we want to be able to make changes to it. Review isn't just so that other people can read your code. It's also so that they can have suggestions and we can think of ways to improve it. It should always be safe to push changes to it. That should never cause anyone to ... if they've reviewed my file and now they don't know exactly what they have reviewed and what they haven't reviewed, that's sad. We want to solve that problem. And we also want to be able to keep up with upstream, either via rebasing or via merging changes back in. We don't want to put that on hold in the middle of a review.

These are the constraints that we want to solve. We solved this in

A very high level summary of it is, if you review something, Iron remembers that you reviewed it. Which is a stupid statement to make, because it seems weird that all tools don't do this. And they do in ways, where if you're reading commits, most tools will show you unread commits, "You have seen this. You haven't seen that." It remembers what you have reviewed, but it doesn't actually remember the content of what you've reviewed.

That's a big difference in Iron's approach, is that the brain remembers explicitly what you've done. Part of this is that there's this explicit action. As you're reviewing you have to say, "I accept this change into my brain." But in the end, it means that you don't have to read code again.

The list of constraints that we've laid out, how many of these constraints are satisfied by brains? Well, actually, all of them, which is cool. All you have to do is remember what you've reviewed, and then you actually satisfy all of these constraints. And this is a weird thing. It seems like such a simple idea, just store this diff and suddenly you solved the problem of merging and the problem of rebasing at the same time. But if you think about it, it's kind of obvious that this is sufficient, because a branch is just this diff of all of the changes that it's going to make. And your understanding of the branch is what you know of that diff. And as soon as the thing that you have reviewed is equal to the changes that have been made, then you have fully reviewed a branch. So mechanically, that's what that is.

But, there's a complicated case. Most of the time, that's true. If you're reading changes one at a time, you're just applying patches to your brain, and eventually it is going to resemble the branch. But what happens if you merge or rebase, such that you incorporate other changes? There's a merge conflict resolution. You don't want to read the entire thing. What you do is something stranger, which is that, you have the diff that is your brain, which is the state of the

otherwise we call it in the future.

You flipped the logic, or the person who made the branch flipped the logic. They said, "I think it's more clear if we ask, if now is later than the current time, than ask if the target time is in the past." So it's fine. You reviewed it. This is your brain.

And then this upstream change happened, where someone came in and they're like Target's a bad name. We're going to call that Target Time. And we're also going to get rid of this anonymous function situation and use this point-free bind style because, whatever. Now, this is weird. You have seen a change. You can predict the rebase conflict that's going to occur. You reviewed a change in this branch that did this. This happened upstream. You know that the two lines that you both changed are going to collide.

And what does that look like? Well, it looks like this, which is alarming, but we're going to walk through it. It's not really that bad. What you have here is a diff of two different diffs. The lines on the far left show you ... These are the diff level lines, and then these are the code level lines. You can ignore these, because these are just context. You can see that there is no material change to the code here. This was a context line. Now, it's not.

These are the only things you actually have to look at. Previously, you were aware of this diff, which just flips the two arguments. And now you are aware of this new diff, which flips the two arguments again, but they have different names. That's kind of awful. That was the simplest possible merge conflict that we could review, the simplest diff that we can turn into something else. And it's already not obvious how to look at that. It's a weird new thing to read.

And it's also weird that, normally we're just reading diffs but then occasionally sometimes we have to read these bizarre, much harder to read diffs. But, theoretically, what is happening is a little bit simpler, where really all we're doing is diffing diffs to one another, but most of the time it's completely trivial. When you're reading a

actual conflict changes that have happened.

And this idea, to be self-indulgent for a second, it kind of reminds me of functional programming in a way, where generally we would think about these references to commits, like what commits have we read. And then we're getting away from that reference mentality in thinking about the values. These are the values that we have actually reviewed, and remembering references to things. We're actually storing the content themselves which you see a lot in functional programming.

There's some downsides to this. In particular, you have to diff an entire branch at once. In the beginning, you can get away with reading a single commit at a time, but as soon as something is rebased it doesn't really make sense. You've seen the future. You can't really go back after the rebase and read one thing at a time. You have to consider the diff of the entire branch, which is annoying and something that at Jane Street we universally do. And we make very small branches. I don't know if there's an elegant solution to this that allows you to look at subsets of the content after a rebase, but I would be interested in seeing if anyone has any ideas there.

Also, no one likes reading d-diffs. Anytime you open your code review panel, and it's, "Oh, well, there's a d-diff now." Then you go through this mental anguish of, "Do I try to understand the d-diff or do I just remove this diff from my brain and start over and just re-review it?" A lot of the time the answer is, oh, this is a trivial d-diff. It's just that context has changed. Or this used to be that, and now it's that. And you can read it very easily. And other times, you can throw it away and just review it again.

But there are a lot of advantages to this. You can rebase at any time, because anyone who is in the middle of reviewing your branch, they have their brain. And when you rebase, they have this new target brain. And the two are diffed, and they always see the same consistent view. It doesn't matter how they got to their brain or

The other thing is that, if you merged changes in, it's the same. You've applied a diff to your branch, the diff that someone is aware of didn't contain the changes from the merge commit and now it does. You're not diffing the entire state of the world from the last time you looked at it to the new state after the merge. You're only diffing the diff that you sought to the diff that is now there. You can see the full contents of a merge, and if there was a conflict, you can review the resolution, which is great. Let me take a drink.

Okay. That's how we read code at Jane Street, but that's only half the picture. Code review, part of the value comes from sharing knowledge and reducing the bus factor and getting someone else's eyes on your code. Maybe psychologically it makes you do a better job because you're worried about what they'll think of you. But, really the value of code review is, we want to make improvements to the code. We want people to look at our code and have meaningful feedback that we can incorporate. And use it to improve, and we can all get better.

How do we do this? There's another really weird thing, which is, inline code comments. This is a thing that is more alarming to me at Jane Street than the idea of using a custom code review tool. The specific mechanism that you give feedback during a review, is that you check out someone else's branch, and you add comments or use the special syntax to indicate this is a CR comment. And then you push back to their branch with these new comments. This is such a weird idea. It felt so dirty to me, but I'm going to explain this to you. And then try to justify why it's really cool, because I found that I really like it.

The current state of the world is kind of weird, too, where generally what I'm used to is, when I make a comment, I'm looking at some diff online and I'm adding the comments in this text area. And then I'm hitting save. Either I'm commenting on the diff or I'm commenting on the branch, or maybe the file at a current state. There are all

JavaScript code. This is how it looks. This is actually what we do. It says CR, followed by the person writing the CR and then the person that the CR is targeted to. And they commit that and they push that. And then, I will see it, because I am also reviewing my branch. I see in my brand it no longer matching the branch of state. I can see that there are new comments. So I can go back in and I can change the CR to an XCR, which is, I've replied to it and assigns it back to the person who originally opened it. And I can either address and correct it or sass.

This feels weird still, but where else should comments about code go? If they're on a commit, the commit can go away. If they're on a piece of diff, then you have this weird heuristic in your code review tool, where it has to try to figure out, it was on diff that kind of looked like this but the new state of the world kind of looks like that. So I think that this comment is supposed to be here now. And sometimes that can fall down. You can also just put it on a branch and say this is associated with that branch, but then you have to put in the comment, "I'm referring to the code on line X of this." It's kind of annoying.

Putting in the code is not as insane as it sounds, and it works really well in practice, with a lot of caveats. Because, this is definitely a time where things are getting worse. If you've just instituted this as a way of attaching comments to code review, your life would be worse.

For starters, your commit history is weird and ugly and you have all these CR comments attached to code or in their own commits. And that's kind of weird, and it's maybe not as bad when you realize by the end of the review they'll all be gone. So it won't affect blame history, but still you know that the graph has these commits that don't really belong there.

But, more so than that, it's really physically painful. You're reviewing someone's branch. You have to go and pull their changes, switch

of the code that you were looking at in the diff. And then you write your comments in your editor. There's almost no context switch. There's just a button, "Take me to the code, because I have something to say about it."

And there are various reasons why this is possible. One of them is workspaces, which we'll talk about a little bit later. The mechanical horror of having to do code review this way, is alleviated a lot by writing a client in Emacs that can do it.

I'm going to talk a little bit about nice things, that I wouldn't really have predicted from this. But actually using it in practice, you see all these advantages to this approach. Which, first of all, there's no context switch between when you fix the code. You can also update the comment and say, "Oh, I fixed the code." You can reply. It's not a two-step thing of, "Oh, I pushed these commits and I have to go back in and scroll down and find where the comment was, and say that I addressed it." You just put it right there.

But, even better than that is, if you're reviewing someone's code and you have something to say about it; if you spot a typo or this variable name could be a little bit clearer, you don't need to write that as a comment. You're already there. You're in the code. You can just make the change. If you spot a typo, you can fix it, and you remove this developer back and forth for, I have this thing, I fixed this thing, oh, I see that you fixed this thing. If you just fix it, then that's not a problem.

Obviously, context is preserved across rebase, which is great, but not that great. More than that, it makes coding more collaborative. When your reviewer is making changes to your branch, there's nice things that happen. When it's less work to fix something and to point out that it's fixed, it's nice. And I want to go into a little bit more detail about that.

When I'm working on a branch, it sort of feels like this is my feature. I am making this thing. I want it to be the best. And when other

Another thing that's nice about this is, you are also reviewing your branch. As other people are making changes, you're reviewing them, as well. Presumably, you have started reviewing your branch before you've given it to other people to review. And then you can see the changes that they've made because your brain will no longer match with the branch.

Some downsides to this approach? None. Yeah. It's fine. There are a lot of weird properties, like the messy history. Although it doesn't affect blaming, it's still weird, and there are things that you could do to alleviate the weirdness. You could just always do it in a separate comment and then make sure that you rebase those comments away once you're done with the review. If you're merging, then there's no hope for you.

But Jane Street takes this attitude that I found very strange. It was, we don't care. We're going to have this really messy history, and it's going to have all these commits in it that just change one thing. Or add a comment, and that's fine. It was a weird adjustment but I kind of like it.

All right. I'm going to talk about a few other features of Iron, and this is a less cohesive chapter. There's this meme that I've heard, that Git is a very good content addressable data store, but it's a really bad revision control system. And it's neat to see a revision control system that was designed for that purpose. What features would you add if you were just trying to optimize for revision control? So I'm going to talk a little bit about that but also still a little more on how we do review.

Obligations: Who is actually reviewing code? It is something interesting that we do at Jane Street. Obviously, if you write a branch, you are going to review your own branch, and before you subject it on the rest of the world. Make sure you didn't make any dumb mistakes, polish it up one last time before you send it to someone else. Then you can add people to the branch as whole

intent of the code in there to the modules. And any change that is happening, if you have review obligations on file, you have to see it. And you get to have something to say about it.

After a feature has been seconded, the review widens out to any files that have been modified. The people who review that get to have a look, too. And file owners, which I'll talk a little bit about later.

The way that we do this is, also strange and intrusive, which is that there are files that are checked into version control. They look like this, as expressions, and they declare some metadata about files. The owner of the file, which is roughly who you should talk to if you have a question about this file. But there are also mechanical implications of that. The file scrutiny, which roughly defines how many people to review it, how many people need to have obligations on it. If it's a very important file, we wouldn't want just one person to have review on it. We might want more than one person to know enough about the file to be able to be a file reviewer.

And you do this by file, by directory. There's easy ways to apply to a group of files. And assign review. As you can see here, we have these n/k review obligations. You can say, "Well, someone needs to look at it from this group," but not necessarily everyone. Obligations, that's how we decide who is reviewing.

Then nested branches. This isn't an Iron feature explicitly. This is just an idea, which is that you can make a branch and then you might make another branch. But it depends on what you did in the other branch. So, instead of just having Master and branching features off of it, you have features that branch off of one another. It's something that comes up emergently all the time. You've probably done this. But it's kind of painful in vanilla git to do this. There's no first class support for nested branches. So, I'm going to show you how we do it and then talk a little bit about what it means for code review.

You have a master branch, and then you create a feature off of it, to

to review this chain of branches that you have, it was this very painful manual, "Oh, well, this needs to be rebased onto that, and then this needs to be rebased onto that." And then you have to go insane shell script to keep everything up to date.

But it's kind of necessary, the way that we do review, to have this, because the atomic unit of thing that is reviewed is not a commit. It is an entire branch. Where you might normally just make a branch that has multiple commits and you read the commits in order and you have this nice convincing story, "Oh, these are good changes. I'll sign off on that." At Jane Street we actually make separate branches and we base them on one another, and you review one branch at a time, just an abstraction layer. This and some other features blur the line at Jane Street between commits and branches. What is traditionally seen as a commit, we treat as branches. Implement as branches, rather.

Another nice thing about this is, we use a mono repo at Jane Street, and we have these permanent features off of that mono repo. They're master app name. There's a master "fe" for all of the Iron features, and those go under it, which is a nice place to release features into. Merge them into this sub master, that is the master specific to an individual project, before you release it to the rest of the world.

Next thing that we do a little bit differently is branch metadata. In git, a branch is a name. It's just a pointer to a commit. We make it this heavier weight thing. We actually call them features instead of branches, and they're implemented as Mercurial bookmarks. But they have this metadata attached to them, a description of the branch. And this is another time where we blur the line, where a commit message is really the branch description. And then branches also have metadata of owner. I used to have this script that would print out all the branches that I had in my repo and then it would print out the name of the person who had most recently pushed to that branch. It's a rough approximation of whose branches are what

and where you are with them.

Up here is review. You can see how many CRs I have, how many people have responded to my CRs. I need to go back. The actual lines of diff that I need to read, follow review, we won't talk about. And then, if there's anything for me to do, I have read this entire file. There are no more CRs. I should second this branch, probably, or go in and add some comments.

It also lists out the unclean workspaces, which is really useful. Because we use workspaces, which I haven't talked too much about. You don't have a single checkout of the branch. You actually have these multiple checkouts and they can all be dirty at the same time... It's like get work trees, if you're familiar with that feature. We use that as a way to quickly switch between branches without interrupting whatever you're doing. You don't need to shelve your changes in order to switch.

But it's kind of weird because then you have this deep directory of every branch you've ever looked at. And it's nice to see it in one place. Well, you shelve some changes, you made some changes here, you forgot to commit them. Don't forget that you have this on your computer somewhere. So that's nice. And then, a list of features that I own in the hierarchy that they are in, pared extremely down so that it fits here.

But it includes the next step, which is things like rebase. This branch has fallen behind its parent. You should probably rebase it. Other things are, this feature has changes, It's fully up to date. You should enable review. Or, this feature has CRs on it. And then finally, CR Soons, which are CRs attached to files. I won't explain that. That's complicated.

This dashboard of things that you have to do isn't novel or really related to Iron at all, but it really changes the way that I go about writing code. It's very easy to create a bunch of different features at the same time, and work on a little bit of it here, and then move

is, rebasing these branches. I have several branches here that need to be rebased that have fallen behind. I can just do that by hitting a button from this dashboard world that is actually an Emacs buffer. And then the rebase will happen in the background asynchronously. And later, if I come to my TODO, it will tell me if the rebase failed. If there was conflicts I need to resolve, it'll show up in my TODO. You rebased this and I was not able to figure it out, so you have to step in. But I don't have to sit there and wait for the git rebase to replay all the commits. I can just get that immediately.

Workspaces: Something that is not at all specific, but necessary for us to do review the way that we do, when we actually are making changes to each other's code. Instead of switching branches, you just have a different working directory per branch. This means that you can do review with less of a context switch. It's also way faster. You just have every branch available in your file system. You don't need to change your entire working directory, which is very important when you have this mono rerpo of massive. Massive.

It's also really useful if you're writing OCaml, because you have these long running build servers that are incrementally rebuilding code. I don't know if that is as common of a problem but it's really valuable to be able to have multiple of these going at the same time. You don't need to restart your build every time you switch to another branch. You can just run all of them at once.

And you never need to stash, which is really nice. This aggressively worded bullet. That never comes up. You still do it to pull and stuff but yeah, it's a big time saver.

Locks: This is a weird feature of Iron. When I think of locks, I think of Microsoft Team Foundation server or Borland StarTeam, or even SVN. This weird idea that I'm going to lock a file, and no one else is going to touch it because I'm touching this file right now. And we use locks in a very different way. But we are still a centralized version control system, which is kind of weird in 2017. But it's really

that. You can also specifically lock yourself to an old version, if you're making a patch on something that is deployed to production. You can create a branch off that and lock it so that you won't accidentally pull on any ... Rebase it and then rebase on something newer.

Locks: It goes with another feature being centralized, is that it's opinionated about what you can do with your commits. If you haven't fully reviewed a branch, you can't release it. It won't let you, because it's a centralized server. It will not accept the operation that you're doing. It can see that people still have review to do. It will not let you release it if you release a feature that has CRs in it. It deliberately stops and forces you to do the right thing, in various ways.

The general nature of this centralization is that, things that change the structure of branches, like rebasing or renaming something, rebasing it to a different part of the tree, creating new branches even, which is kind of annoying. There's this compromise between safety and productivity. You can't just do whatever you want and everything will work out fine. And you don't have to worry about the force push problem. And you are still using Mercurial, so you can still work locally without ever interacting with the server and just not release your changes on the rest of the world.

Okay. We talked a lot about Iron, which we use as a code review tool, but once we had this tool, we started using it for a lot more than that. We review source code, obviously, but we also put configs in Iron. And we put obligations on them, such as the people who understand when these things could change, can see any change that is being made to them. And then, we go through the same normal review process that you would do for a code. We have a blog and we review our blog posts. Using it as, if you write a new blog post, you're going to add a reviewer to it. They can read over it and give you comments in the normal way.

reviews. You want to apply it more places like, "What else can we review? What else can we give ourselves file obligations on?"

That's all I'm going to say about Iron. Hopefully, some of these ideas struck you as positive insanity. Maybe even such that you will go out and reproduce them so I can use them when I'm doing things outside of Jane Street. Make an alternative implementation that is not OCaml, so other people can read it.

You sat through a fairly long talk about the mechanics of doing code review and how we do that. So you probably think a lot about developer tools, and this is how you're choosing to spend your Wednesday. If you're interested in coming to work somewhere that spends a lot of time also thinking about these, considering applying at Jane Street. You'll have to write inline code comments but you'll slowly Stockholm Syndrome-ly learn to really enjoy it. And then give a talk about how cool it is to someone else.

So, we have plenty of time for questions. I might be able to answer some of them.

What's the difference between a nested branch and just a branching off of each other on the machine?

Nothing. It is not really an explicit feature. There are tools that make it very easy and that make it well supported, but it is exactly that. It is just a formalized mechanism. The way that I would implement this is just, name a branch in the same nested way that we do, using slash separates. And then write a shell script that automatically rebases things such that they're in the right place. There is not really a different feature there.

Yeah?

I have never used Mercurial, and most of the people I know haven't used it in several years. So, this is a question coming out of ignorance. How much of the flavor and velocity that you like in Iron, is due to the fact that it's running on top of the Mercurial as

Otherwise, they're really very similar. Ever since Mercurial got bookmarks, the philosophy of Mercurial hating rebasing is the only difference that I'm aware of in the future sets. And Git Worktrees is a great feature that Mercurial lacks, but we reimplement that in Iron, so we still have that.

Yeah?

I wonder, do you find that placing the focus on the level of small branches, versus commits frees you to make commits in a way that you're using them as a tool to develop faster because they're not seen as a record of your style as a developer in a way that you might want to say squash them so you don't appear incompetent?

That's a great question. And yes, absolutely. When I first started at Jane Street, this was very weird to me, the fact that commits don't really matter. I'm used to writing long, informative commit messages. And before putting a branch in review, I would rebase it several times until I was happy that it expressed... I would reorder things and do all these things. I don't do any of that anymore, at all, ever. I did like doing that in a weird craft of software development. I felt good about the branch, but it doesn't actually serve anything. So, divorcing the code from the commit history entirely, I still don't do it outside of Jane Street, but it doesn't bother me at all anymore.

We even have this setting that is not enabled by default, but that almost everyone enables, which is, to not write commit messages at all. There's a button, and it is commit, and there is a single string, and that would be the commit message. I use underscore now. It's this weird thing that made me very uncomfortable, but as soon as I started doing it, I don't have to care about this. I make a change, and commit it, and then go around and do something else and I commit it. And then maybe go and undo the first thing, commit it. And I'll just push all those commits. It's horrible if you think about it, but it's fine. Yeah.

we can just roll it back.

There's actually another interesting thing that we do. And this is not directly review-related, but there's another feature of Iron that I didn't talk about called Traits, which is, if you find a bug at a given point in time, you can create a trait, which is, this commit has this bug. And then because we never rebase, a nice actual property of always merging and never rebasing, anything that is an ancestor of the commit that you have observed to the bug on, knows that it is also infected with this trait. If you spot a bug in production, you tag the commit that you saw it, and then when you fix the bug, you say it was fixed at this point.

And then, for any given revision you can ask, "What bugs does this revision have? What has this not incorporated?" We would roll back to actual executable, and then make the traits and then make sure that when we rebuilt it, we've rebuilt it in such a way that there are no more buggy traits.

Yeah, but we wouldn't back out the commit or go back and amend it and fix it. We would just make a new thing and change all the code. Or possibly even do the HG backout thing where you just apply the reverse diff and then commit and push it. Which is also how we remove a commit from a branch. And we don't actually rebase and get rid of that commit. We'll just apply the reverse and push it to the dirty history.

Do you ever do the equivalent of git bisect? And if so, how do you handle the messy commits, and how do you understand where they are? Or maybe the answer is, you just don't do it?

Yeah.

Can you repeat the question, please?

Bisecting. Bisecting is this great, wonderful, debugging feature, where you essentially ... If you're not familiar with bisecting, what you do is, you pick a revision that you know to be okay. And then

introduced the bug.

It's super nice and really valuable and we don't have it at all. The fact that we do these constant merges that we call rebases in order to keep up with upstream, means that you can bisect but where you're using Mercurial, it is a tool that is available to you. And it will find the merge commit. And then you bisect, dash dash extend, and you follow that tree. And you find another merge commit. It's extremely painful.

But we keep a history at the feature level. This is not something that I've implemented but something I've often been, "Oh, man, I should really write this," which is to do bisection at the branch level. Only look at the points in time in which you have released a branch into its parent and then bisect them that way. But we don't have that, and it is annoying.

Yeah?

You hit about this idea of everyone can mess around with your code, everyone [crosstalk 00:54:52]. And you also have file level of owners. Those two things seem to be in conflict.

Yeah. File owners, really the only thing that it means is, if review has finished and a branch has been merged into Master, and there are still CRs in it, then those CRs, instead of being assigned to the branch owner, they will be assigned to the file owner. They're called CR-soons. This isn't an immediate concern, but this is something we should fix soon. So you change a CR to a CR-soon, that makes your feature releasable, and then in your TODO you can see the CR-soons that are assigned to you and the files that they come from.

That's all that file ownership really means. The file review obligations are much more ... Yeah, that is the only thing that being an owner means. Those are the people who understand the file, and generally the owner is also a reviewer.

All right. Great. Around time. Thank you.

Jane Street

you

View open roles