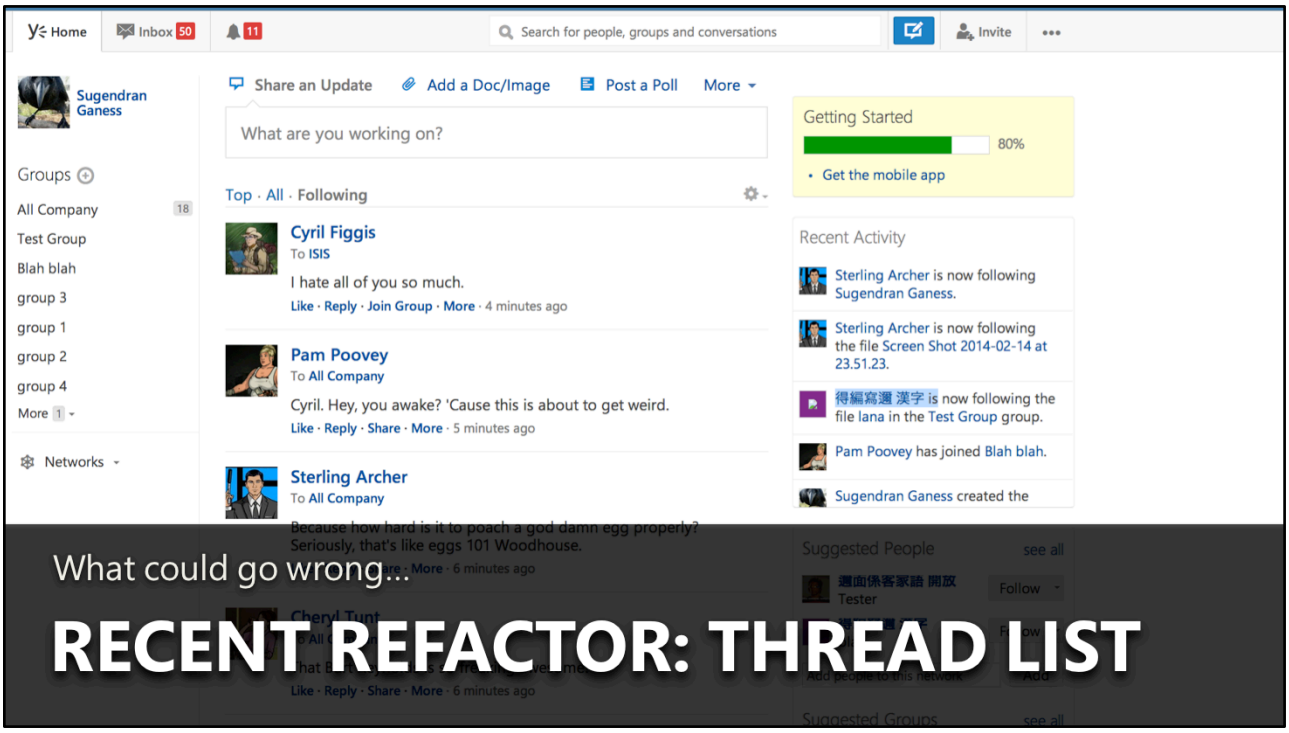
Refactoring Legacy Code

@sugendran

Microsoft

Hi. My name is Sugendran, you can find me on the Internet as Sugendran. With the release of XMLHttpRequest 15 years ago the idea of a dynamic interactive website became very much a reality. Though it wasn't really until the browser war [Firefox vs Internet Explorer] that started 4 years later did I really started to understand what it meant to build a dynamic website. Back then we didn't have mature component systems, so we just rolled our own. A lot of engineers I talk to have similar experiences. That was 11 years ago, so let's fast forward to today. We now have an Internet filled with mature sites that have progressively evolved over the last 5+ years. It's not uncommon these days to join a company and look in horror at the untested, undocumented spaghetti code. The thing to remember though, is that the code, no matter how much you dislike it, does the job it was written for. The real problem is that working in this type of codebase can be quite slow, not to mention difficult to understand. We recently refactored some 4 year old code at Yammer, and I'm going to talk about how it went.

The Yammer frontend repository is quite a busy place. We've got 26 frontend engineers spread across three offices, London, San Francisco and Seattle. Right now we're averaging about 30 commits to master everyday. Seems like a lot right? It's because we have a combination of bug fixes and feature work that we ship everyday. If you haven't heard about Yammer, we're the corporate social network and we approach engineering in a slightly different manner to our enterprise counterparts. Our engineering team takes requirements from the product team and we work together to deliver a testable MVP. The MVP ships and we look at changes in our core metrics to decide if a feature will live or die. These smaller MVPs means we're always adding and removing code. With over 4 years of engineers getting their grubby fingers into the code we end up with a mix of custom frameworks and open source libraries. Not cool, right? Something the team has recognised is that for anyone jumping into this codebase the learning curve can be quite steep. Our solution is that overtime we are going to refactor everything to use a common M-V-Something framework. We picked Backbone.js for it's simplicity and the fact that it wasn't too pushy about the right way to do something. That



Myself and another colleague were on a project to refactor the ThreadList component. The ThreadList component powers the feed on Yammer. The feed is a list of threads each with a thread starter and some replies. It's used to surface relevant information to the end user and is the core experience for Yammer. It's also where we've done the most amount of experimentation over time. And time does take its toll on a codebase. No matter how carefully you refactor code, there is always going to be a trail left behind. With 4 years of experimentation it has made this part of the codebase extremely hard to follow. The rendering code was a complex sequence of functions that were called asynchronously. Most of this was written to give the impression of speed in slower browsers that are no longer supported. The biggest problem with asynchronously building the feed is that there is a sequence of events that is not very easily understood. This meant that anyone jumping into this part of the codebase would first have to spend quite a lot of time to understand what was going on before they could get on with the task at hand. Now imagine having to do this every time you switch context back to the thread list – this slows your team right down.

Before we get into the refactor it is important to remember that no matter how much you disagree with past decisions, they did the job they needed to. It's very possible that if those decisions hadn't been made then your current employment situation would be very different.

Rewrite vs Refactor

- DANGERZONE
- 4 years of changing functionality
- 4 years of bug fixing
- Interdependent code

Now, usually when code gets this complex and difficult to work in, we as engineers end up having a rewrite vs refactor conversation. It's a tricky decision and the answer isn't always the same for every business. For Yammer the decision is very simple, rewriting would mean we would pause moving the product forward while the rewrite occurs. For Yammer the time to ship is a lot more important during a rewrite than it is during a refactor, as us engineers should not block the product team from moving the product forward.

We've got 4 years of changes in functionality and bug fixes. This makes for a significant piece of work. And, before you say that we could rewrite in parallel, that would mean a significant compliance cost to the team doing the rewrite. They would need to make sure they're aware of all ongoing experimentation on the feed and integrate as appropriate. And the longer they spend on the rewrite the more integration they're going to have to do. So for us we're pretty much always going to refactor because we can do it in an iterative approach, refactoring in smaller amounts to get us to a better place.

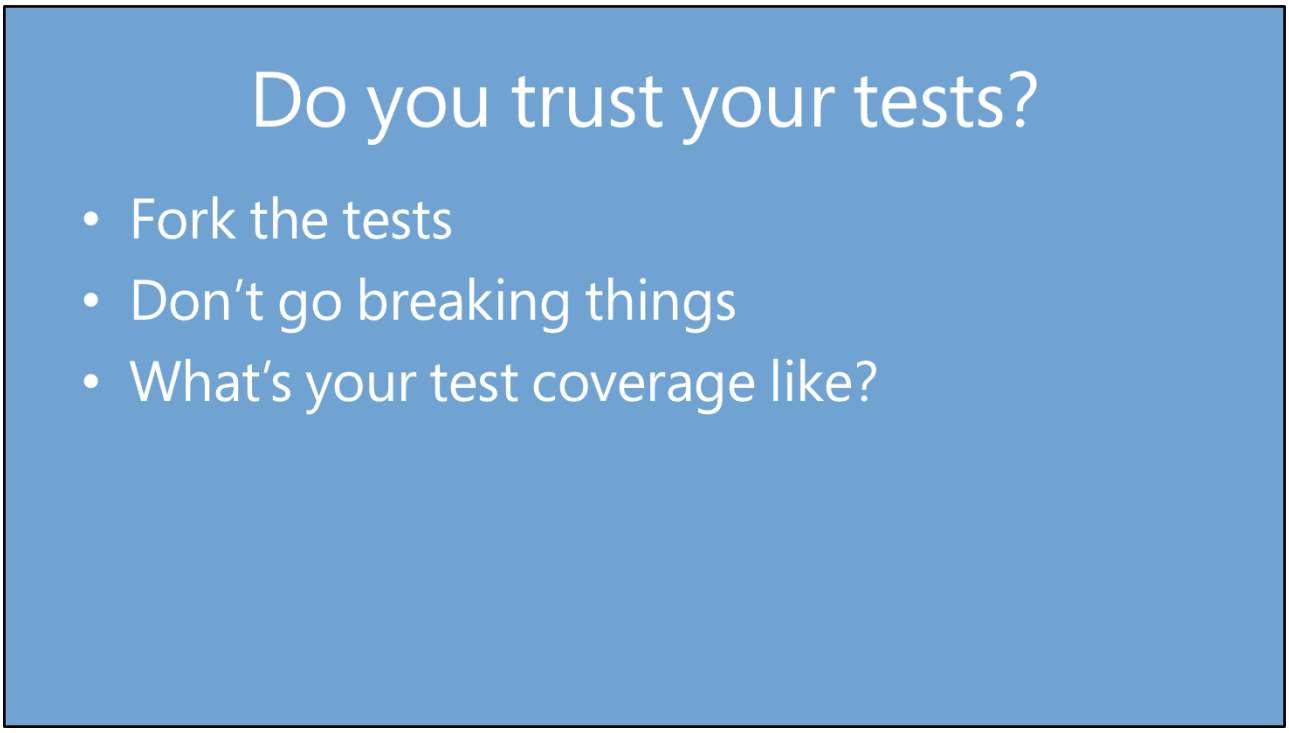
One of the more interesting rules at Yammer is that nobody works alone. It's a rule that I believe in. I'm pretty sure that I don't have all the answers. I certainly have some ideas, but it's quite easy to miss things. Having someone to bounce an idea off helps. Especially so when your colleague is as stubborn as you are. So let's dig into the steps we took.

Understanding the Domain

- What are the use cases of this component
  - Feature Specifications
  - Bug fixes over time
- What depends on this component
  - Code
  - Tests

First up, do you fully understand the domain you're working in? I thought I understood what the threadlist did when I started this refactor, but it wasn't until we did our research did we work out the full extent. I can't stress enough that before you start to code you must first properly understand the full scope of your project. Odds on the functionality has changed over time and it is not completely documented. We spent a full week doing our research. We wanted to make sure that we had a clear picture of what pieces of code depended on the components were we going to refactor. We also used this time to limit our scope, as it was very apparent that large parts of the website either inherit from thread list or reference the thread list.

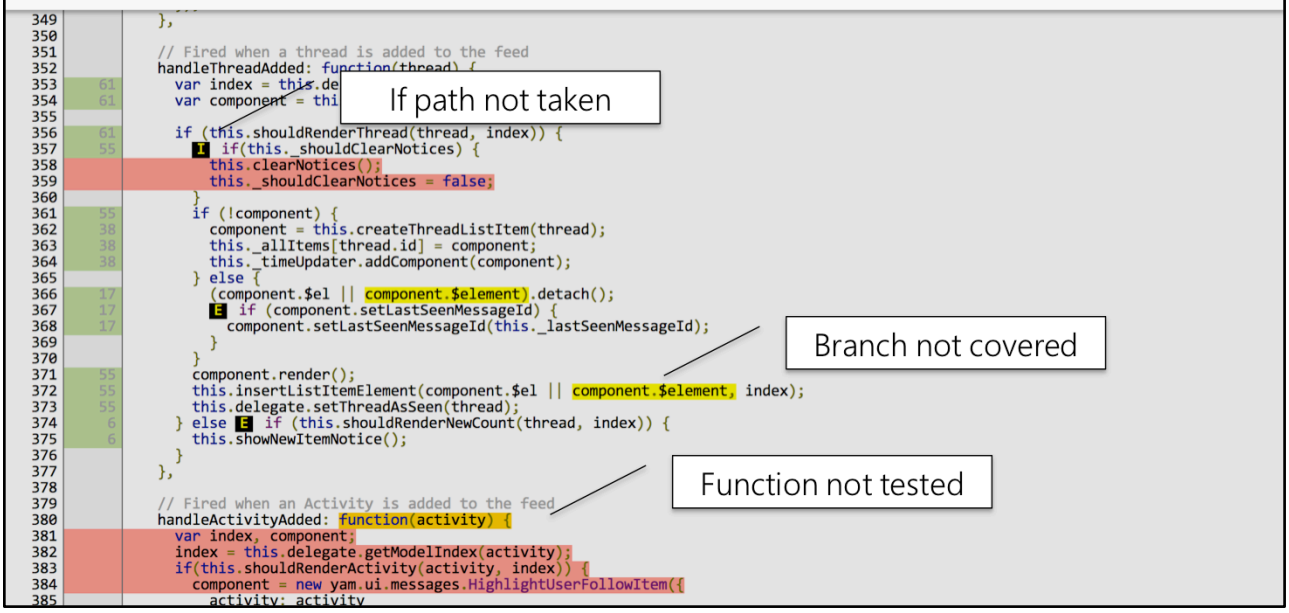
And, The first thing we did after our research was to commit some code to master. Yep that's right shoved some code straight into master. We copied and pasted the thread list class, renamed it and stuck it behind a feature gate. A feature gate or feature flag is simply a mechanism that only enables a feature for a specific set of users. At Yammer we have an AB test system that let's us roll out features to a subset of users. Aside from letting us test new features it also let's us test code changes without effecting the whole user base. I know not everyone has an AB test system, in the past I've worked at places where we've hard coded our user ids in an if statement. It's not pretty but it does get the job done. Using the feature gate means that we don't have a long running branch. Instead we do smaller iterations and frequently merge back into master. This is really important in a repository as busy as the Yammer frontend. During our refactor we had 3 projects do work in the feed. If we were in a separate branch we would not have been aware of each other's changes. Since the rest of the team can see your changes sooner this helps team aware of what's happening. Our first merge back into master happened after a month, and then after that we were merging



Along with copy and pasting the thread list class we also copied and pasted the tests. Since end user functionality was not going to change most of the tests still applied, but we were pretty certain that some of it was going to change. Forking the tests meant that we could have some confidence about our refactor without leaking changes into the existing codebase. I can't stress enough that both versions of the component should be under test so that we DON'T GO BREAKING THINGS.
It's also important to note that the original thread list didn't have any test coverage to speak of. Over the last two years we've managed to get it to a point where we're happy with it – about 70% coverage with our unit tests. If you are trying to refactor untested legacy code without first adding tests to it then you might as well be rewriting it. It's only by adding tests that you can be confident that all existing functionality is covered. By starting from a place where we trust our tests meant that we were pretty confident we were going to not run into too many unknowns with our refactor.

Istanbul

http://gotwarlost.github.io/istanbul/



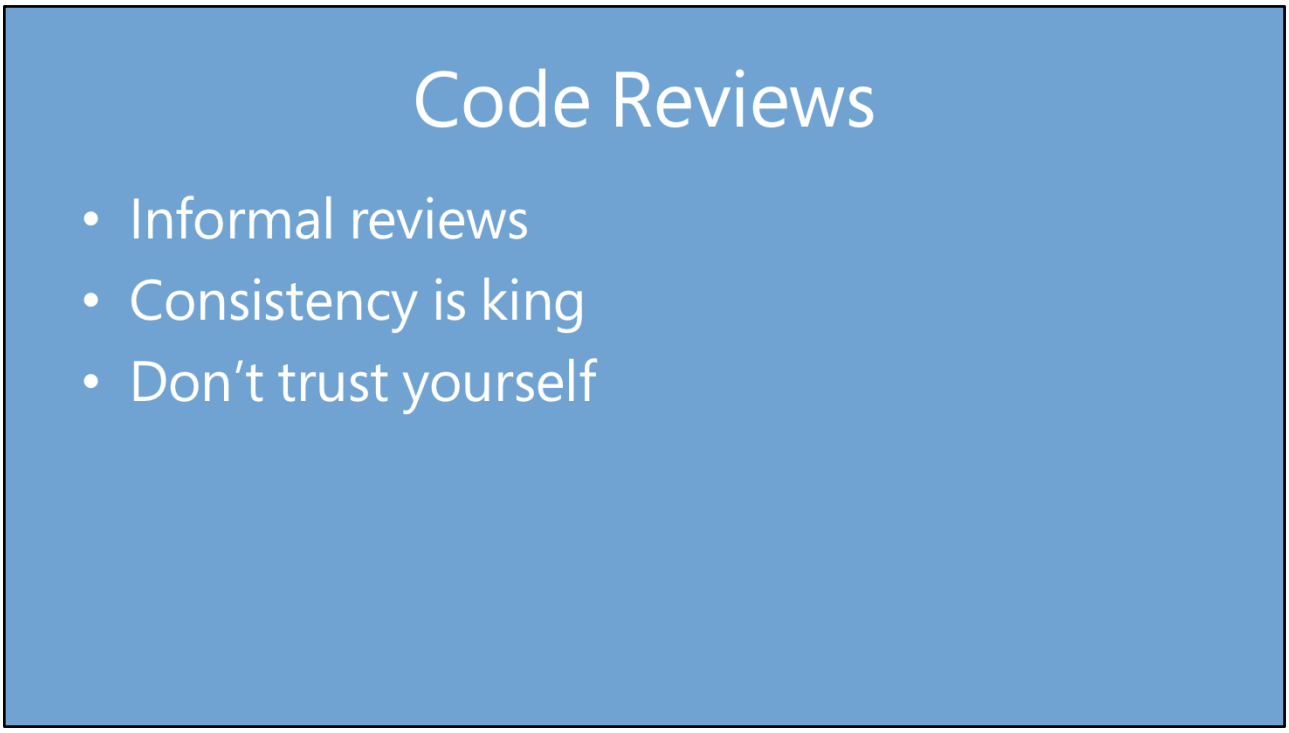
One tool that helped get us there is called Instanbul. It's a pretty neat code coverage tool that instruments your code and tells you which code paths are not covered. The attached screenshot shows where we could improve our tests. There are conditions and statements that don't get tested during our unit tests. By looking at the test coverage report and progressively adding sensible unit tests we're able to get our coverage up.

Think about the API

- How will people consume the new code
  - Inheritance
  - Events
- Be explicit
- How did we get to this place?
  - Reduce complexity
  - Don't prematurely abstract

With the feature gate in place and the tests copied we spent some time thinking about the API of this component. This mostly meant the two of us in a room with a whiteboard arguing for two weeks. In between the discussions we hacked up branches with the various approaches we had in our heads. They weren't complete solutions but it gave us a good idea about how feasible the approach would be.
The reason it took us so long was we wanted to make sure thought about the API properly. We also wanted to make sure we were very explicit about how the component works and how it can be extended. For us it was important to be explicit. Our refactor was because no one understood how the abstraction built on top of the abstraction actually worked. So our guiding principles was to be explicit about how the feed rendered, and not to abstract something because we think it might be useful elsewhere.

One tool that helped us track the complexity and maintainability is Plato. Plato is a tool that runs against your codebase and tracks complexity. We use this tool to periodically make sure we were making the codebase easier to understand. Complexity is a metric that measures how many paths your piece of code takes, so the idea is that if your code tries to do too much then it's more likely you're going to introduce unexpected behaviours. By splitting out functions to reduce complexity you end having smaller functions that are explicit about what they're doing.



Even after we worked out the API and started writing code we would continuously ask each other to double check the code getting merged in. Since we were merging every few days, we ended up doing an informal review every two days. They were pretty short and usually went along the lines of 'Hey, can you check out branch X there are some changes I want to merge into master." I found these informal reviews to extremely valuable. They weren't about picking apart the style problems, it was about making sure that the code made sense to someone else. This is really important at Yammer because no one person is the gatekeeper to a piece of code, instead we work across the codebase as the project requires. So if I'm the only person that understands the code I've written then I'm going to block everyone else. The reason these reviews were informal is that at Yammer each team runs the project as they see fit. We didn't have a formal code review phase, instead we opted to progressively do it when we felt it was needed. In hindsight I think this was a good approach as we were both aware of when our code should be reviewed. And since each review was small and iterative didn't need to interrupt each other's flow with larger pauses for reviews. I imagine with a larger team we would have considered a more formal review process so that everyone on the refactor was aware of what was going on.

Constant Communication

- Keep the team in the loop
- Reduce future shock

Regular informal code reviews made sure the we kept on top of what each other was doing. However communicating this back to the larger team was much more important. You see the most important thing about changing code is not rewriting all the code, rather it's having the rest of the team accept the that the changes are for the better. The last thing you want is to ship a major change and have the rest of the team resent you for making their life complicated. To make sure this wasn't the case we did a few things. Before we started working on the refactor we got feedback from the team about what was they thought was the problem with the thread list. Once we had this data we put together the tech spec and shared this with the team. And then as project went on we made ourselves available to the team to talk about how the refactor would effect their projects.

When is the project finished?

- Remove the feature gate
- Dust Hands
- Walk away
- …
- Profit

So when do you stop refactoring? Way back at the start of the project we set up the scope. Sure there are other things that you could do while you're in this codebase, but you're going to start seeing diminishing returns if you don't ship. At the start we set out to make the code easier to understand. We feel that we did that. Yes there are plenty of other things that we want to tackle but what's more important is getting the rest of the team to start using the refactor. So we removed the feature gate and let it bake in production for a bit.

Avoid the long tail of bugs

- Release often and find bugs along the way
- Features you didn't know about

We were pretty confident opening it up to production. The feature gate we added way at the start was part of our AB test system. This mean that once we were happy with the first cut we were able to enable the refactored code just for Microsoft, which Yammer is part of. Releasing early gives us time make sure we aren't going to introduce bugs since everyone has a different set of features we often use. An example of this was a link we show at the bottom of the feed to take users back to the top, it's something we never use ourselves. In the first day of pushing this out to Microsoft for dog fooding we got feedback that we'd broken it. Now not everyone is going to have a large company like Microsoft to dogfood the product but you do have other team members, even those guys who aren't engineers. In previous jobs I've used the non-engineering teams as my test users because they just want the product to work and they're a lot less tolerant to features not working.

Future Works

- Phase 2 and 3?
- Notes for future teams

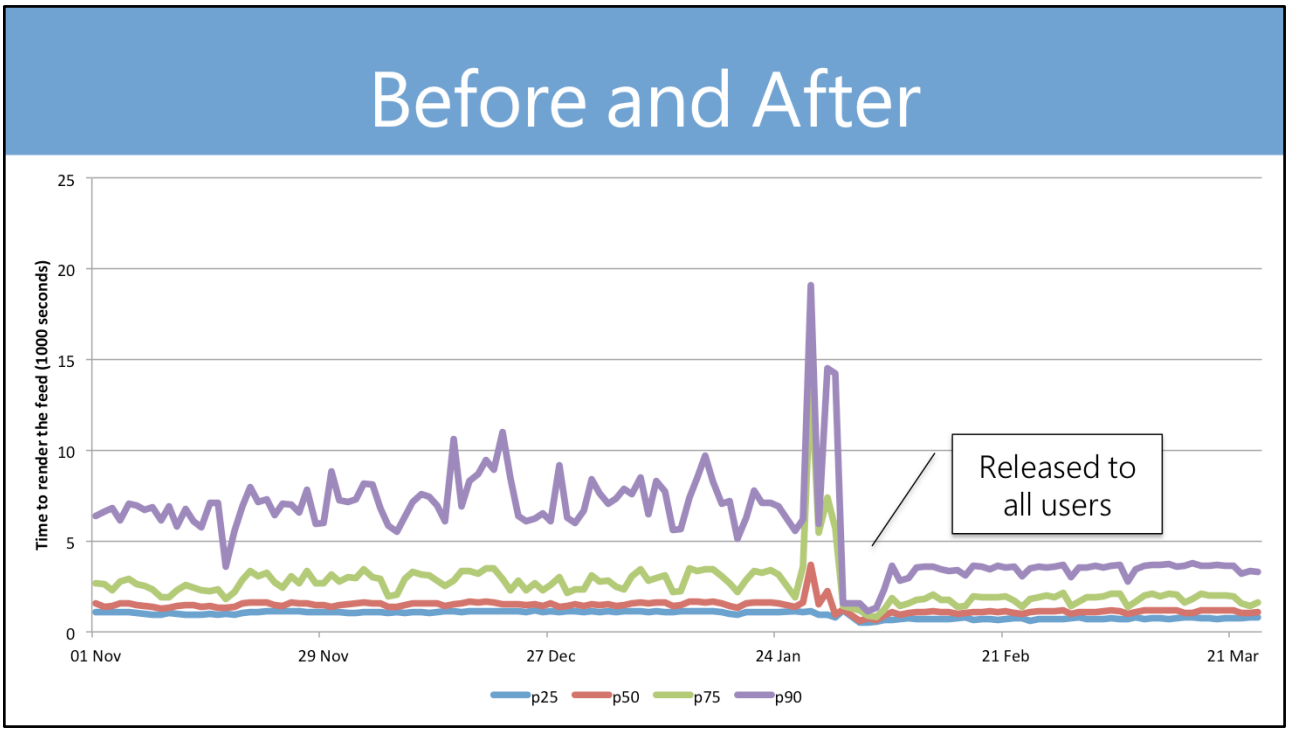
So we shipped, that's it right? Almost. Remember all those times you said "no we can't tackle that because it's out of scope" ? Those are the things you need to make notes about. Just because your project doesn't have scope to tackle it, doesn't mean it shouldn't be tackled. This is your chance to highlight future work so that the business knows there is more work to be done. It's also your chance to give future teams a headstart. Document as much as you can so that future projects don't have to spend as much time as you did scoping out what needs to be refactored.

Before and After

	Before	After
Files	11	5
Source Lines of Code	2454	1456

At the end of it all this is what we ended up with. 40% less code. We removed the async rendering and made it very clear as to how threads are rendered. This resulted in a predictable rendering profile which we optimised as much as we could. The net result was that the user didn't notice a difference in most cases, and in the cases of slower browsers we ended up with quicker rendering. At the start of this talk I said we did the refactor to make it easier to work in. This is something that is hard to measure, but we do have some empirical evidence. There is currently a team that is building an experiment where they have to change the layout of the feed quite dramatically. For them they have refused to work in the old code and are only doing the changes in the refactored Thread List.

Before and After



Aside from making the feed easier to work in we've also had a nice side effect in terms of rendering performance. This graph shows the render times for grouped by percentile rank. For 50% of our users there wasn't a large change in render time. But for the remaining for 50% we were able to not only make rendering faster but also more predictable. The biggest reason for this is that we removed the asynchronous rendering, which would be effected by animations and other components rendering during the feed render. Since the feed is the primary experience we're happy to render it first and then render any other component, which is what happens with our refactor.

All in all we believe this was a successful refactor.

Questions?

Picture sources:

- Footprints - Roger Ferrer Ibáñez
- Window Washers – Kevin Harber
- Traffic – Fygget
- Merge – OneFuller

@sugendran

Thanks.