Thank you for coming out this morning. This talk is going to address the topic of how to make working with legacy code less painful. As the title implies we'll be talking about the use of Bounded Contexts, which should be familiar to anyone who has familiarity with Eric Evans' writings on Domain-Driven Design. To get things started I want to make sure we have some clear definitions.
First, is the term Bounded Contexts. As I’ve already said I am leaning on the meaning of this term as it comes from Domain-Driven Design, but for clarity there is the definition I will be working from.
Domain-Driven Design Distilled defines a Bounded Context generically as [CLICK] “a semantic contextual boundary.” That is a meaningful but terse definition. To expand on what is being described in that, and the text that surrounds it in the book I like to work from the following definition for Bounded Contexts:

“A meaningful collection of components that meets a defined, limited, and useful purpose with defined interfaces and means of interaction with external actors.”

This is still a generic definition, but I think it clarifies certain aspects of the idea and reinforces the “bounded” nature of Bounded Contexts.
The other term I want to have a definition for is Legacy Code. This is a term that can be ambiguous, and for anyone who heard my talk yesterday, I dislike the term “Legacy” as we tend to use it in our industry because I think it is too often used as a pejorative term. But, I think it can be defined in a way that does not reinforce that negative notion, and that is what I want to work from.
“source code inherited from someone else”

— Legacy Code, Wikipedia

Code that because of its lack of tests, poor design, or other factors is difficult to work with.

Wikipedia includes an interpretation of Legacy Code that is humorous, but accurate with regards to how this term is too often used.

The unfortunate reality is that Legacy Code is often a pejorative way to describe code that some developer does not like. The reasoning for this label varies, but it usually comes back to some facet of the code that makes it difficult to work with. For this reason, I like to define Legacy Code this way:

“Code that because of its lack of tests, poor design, or other factors is difficult to work with.”

This does not remove the pejorative tone to the term, but I think it helps move towards identifying what makes the code difficult, rather than just bemoaning it.
When we come into a legacy code base and want to make changes what do we hope to achieve?

1.) Keep the rest of the system working
2.) Improve the overall system by adding value, or reducing debt, or both
3.) Increase the pace that future changes can be made

Other more specific goals are likely to come along as well, but these are the three big goals I have when I start working on a legacy application. I don’t want to rewrite it, I don’t want to get rid of it, I want to make it better.

The goals of doing work in legacy code are not philosophically dissimilar from home renovations. Keep the good, get rid of the problematic, and increase the livability. In the case of code, we want lower maintenance costs, which tends to mean code that we can change quickly and with confidence.
At this point I want to pause and call out some resources that I highly recommend and that have shaped my views on Legacy Code and why I think many of the impulses of our industry around these kinds of projects are less than ideal.

First [CLICK], is “Domain-Driven Design” by Eric Evans. Not only is this where I was first introduced to Bounded Contexts, but the perspective that it is of critical importance for developers to understand the business domain they are working in was crystalized for me by reading this book.

Second [CLICK], is “The Nature of Software Development” by Ron Jeffries which helps focus on the delivery of value when building software. This book helps orient us towards the kind of customer-centric thinking that I think is invaluable as well as keeping oriented towards the quality of the code that we produce as well.

Third [CLICK], I recommend the book “Beyond Legacy Code” by David Bernstein. This book is about process and practices, and reinforces things that underpin the whole agile methodologies movement. Its specific emphasis on improving Legacy Code also makes it worth your time if you work extensively with such code bases.
Finally, I want to recommend a presentation that Simon Brown has given a number of times on the topic of Modular Monoliths. His approach is really compelling and has informed my own thinking on the subject of this talk.
The project I’m going to use as my example for today was a Software-as-a-Service offering from a previous employer that was built as a monolithic Ruby on Rails application. It was under constant development, but certain parts of it were not easy to work with. So, the team I was leading worked to keep the features we needed to add well encapsulated in comparison to the rest of the system and to maintain strong boundaries between our additions and the third-party service we were also integrating with.

The business domain that we were working on was that of a project management tool and the feature we were adding was related to the reviewing of proofs for visual and print media projects. The existing system already had the core model that we needed to build on with regards to projects and stories, as well as the account and user models that would be necessary to integrate with the third-party service. What my team needed to build was the integration layer for the backend, as well as various front-end user workflows for attaching, revising, and approving proofs on individuals stories.
The Problems

The existing system that I'll be describing was developed over several years by dozens of engineers. It had a lot of good things going for it. There were a lot of tests covering acceptance criteria, integration of components, and functional units. The test coverage made most changes manageable because they instilled confidence. The teams working on the project also had good practices when it came to working in small steps, refactoring as we went, and keeping the system well tested as features were added or changed.

But, there were also problems. As a large Rails application there were a lot of models. And, as a Rails project there was very little distinction between the domain model and the data model. For the overwhelming majority of the application they were one and the same. This led to a degree of coupling and interconnectedness across the system that was very difficult to keep track of at times. It also meant that changes would occasionally have completely unexpected side effects because business logic was embedded in lifecycle events and custom validation methods. This also led to the test suite being very slow, to the point that it was no feasible to run the entire test suite locally and instead we relied exclusively on CI for all but the tests written as part of feature development.
Because of these problems the teams I worked with on the development of the feature I will be describing wanted to do things differently. We wanted to keep business logic separate from persistence whenever possible. We wanted to keep details of the third-party service we were integrating with from polluting our model of the domain, and we wanted to minimize the amount of additional behavior we were adding to parts of the system not directly related to our feature, thus limiting the risk for other teams whether they were working on enhancements to our code or things completely unrelated.

These desires led us towards the concept of a Bounded Context. Additionally, since the feature we were working on was an add-on to the main product, we knew there would be a stronger degree of separation that should be maintained for ease of administration.
For simplicity I’m going to focus on the backend work related to the project since that is where the use of a Bounded Context proved the most useful. On the left are the models in the existing system we knew we would need to interact with. Unfortunately, all four of these objects represented God Objects. All of them had lots of behavior that impacted nearly every other part of the existing system.
Like I already mentioned we were also needing to integrate with a third-party service. This service also had its own model for representing some of the same concepts that our existing system had. Because of this we knew we would have to maintain a mapping between our notions of these models and theirs. Thankfully, we were able to ignore certain details of their models that were not relevant to us. Since we were white-labelling the third-party service, we did not need to differentiate the various account and user types their service supported, nor all the various states their models could be in.

What we did need to deal with was that when an account on our side had this integration feature enabled we needed to create an account in the third-party system.
The first thing we introduced was an AntiCorruption Layer that was comprised of two different components.
The AntiCorruption Layer came to have two main subcomponents: a handler for web-hooks allowing the third-party system to inform us of changes on their end. And, our definitions of the various commands and queries we would make of the third-party system.

The web-hooks allowed us to handle the results of asynchronous actions we initiated, such as uploading a file as a proof for processing.

The commands and queries represented all the interactions we had with the system and were split into two groups based on the level of permissions needed to perform them: either partner-level actions such as creating accounts and modifying their state, or account-level actions such as creating users, projects, and proofs.

Both components of the AntiCorruption Layer grew over time, and as we got further into the project the identification of these components as part of the AntiCorruption Layer emerged fairly naturally.
The AntiCorruption Layer provided the necessary bridge between our system and the third-party. The details of the mappings between our system and the third-party belonged to a specialized model we simply referred to as the Mapping model. It held the unique IDs from both systems, details on the model being mapped, and some metadata to facilitate certain asynchronous behaviors.

At first we introduced lifecycle callback logic directly into the models of our existing system to interact with the AntiCorruption Layer. Eventually we extracted the logic of those lifecycle callbacks into Method Objects that were a part of our Bounded Context, but the models in our system maintained awareness of those Method Objects since they were still invoked as part of the lifecycle of those models.
After dealing with the basic housekeeping associated with maintaining the link between our system and the third-party we introduced the new feature concepts to our system in the form of Proofs and Revisions. Proofs belonged to Stories and Revisions belonged to Proofs. On the third-party end the concept of Proofs and Revisions were a little muddled, so we relied on our AntiCorruption Layer to smooth out the interaction for the purposes of how we wanted to model things.

A Proof always had at least one Revision, and the Proof would refer to its latest Revision when asked to provide details on itself. Eventually we exposed the revision history as we expanded the feature set. Now, the interesting thing my team did that distinguished our Bounded Context from the way the rest of the app was constructed related to how we managed these Proofs and Revisions, and eventually other details that I didn’t directly work on but which I advised the team on.
In most of the existing system we had embraced the classic Rails approach of relying on Fat Models for behavior. For our Bounded Context we made a deliberate effort to keep of models, the Proofs and the Revisions especially, focussed on persistence since that is what ActiveRecord, the Rails persistence library, is really good at.

We pulled every bit of functionality we could out into Service Objects that handled individual actions pertaining to these models. This meant that all our business logic was clearly contained and testable to various degrees of isolation. This allowed us to iterate very quickly as we expanded the feature set of our Bounded Context.
Recap

While the application I described was not Legacy Code in many traditional senses it still had certain characteristics that made it unpleasant to work with at times. To address these issues the teams I worked with to developing the products new Proofing feature looked for alternatives to the approaches used within the code already to reach a better outcome. We were able to employ a Bounded Context to isolate the work we were doing to enough of a degree to make our feature development faster and safer.

We relied on an AntiCorruption Layer to help protect our domain model from that of the third-party service we were integrating with. And, while we still had to integrate with multiple God Objects within the existing system we were able to minimize the amount of changes we introduced by isolating our Bounded Context’s business logic to Method Objects and Service Objects which could be changed without changing other objects in the system, and which could be tested in a higher degree of isolation.

The overall outcome was very positive and while we didn’t achieve the level of isolation I wanted in all areas, the team lead, and other team members, I left the work with understood the direction and benefits of the approach undertaken.
Along the way we did encounter some difficulties. Because we were breaking with a lot of established patterns and approaches in the existing code we had some slower code reviews at times. I also completely bypassed the concurrent story of the front-end development of the feature work, which was handled by the same team. After I stepped out of the Team Lead role some parts of the front-end work were undertaken using React, which was a new technology for our front-end development work. This change led to some additional difficulties with relation to code reviews, integration with existing front-end code, and the like.
The team regularly achieved something because of the approaches we took and that was that we regularly ran out of work in our backlog. By taking a different approach, focussing on good separation of concerns, and continuing to maintain the best of the organization’s existing development practices we achieved consistent velocities that outpaced our Product Manager multiple times. So, even with the difficulties we encountered we still were able to perform at a very high level and deliver our features as needed.

We also produced what I felt like was one of the easiest areas of the code to work with. The domain model was well represented, business logic was cleanly implemented, and our impact on the rest of the system was well managed. For me one of the biggest successes with the work I led my team to do was that it was relatively easy to hand-off to another team lead. A lot of this owed to the fact that the team lead that replaced me had worked on a lot of the code with me, but our approach made bringing on new team members much smoother than it would have been otherwise and the way we organized the code allowed other team members to contribute in meaningful ways to the continued development of the feature with high degrees of confidence.
Questions
James Thompson
Staff Software Engineer @ Nav

@plainprogrammer
theplainprogrammer.com