

Anthony Panozzo's presentation notes [22ideastreet.com/blog, panozzaj@gmail.com]

First I wanted to give an overview of the project that I worked on, then give a high-level description of TDD, then share with you my personal takeaways and lessons learned.

So the project that we worked on was a web application for managing the automation of rebates of a pharmaceutical product.

Pharmacies and wholesalers typically stocked and sold this product to customers. However, patients with only Medicare and Medicaid would only reimburse pharmacies half of what they paid for this pharmaceutical product. Obviously this meant that the pharmacy would be taking a loss, so our client used rebates that they would give to the pharmacies to encourage them to add shelf space. While the pharmacy would take a short term loss, it would eventually be compensated by our client. The client was making a huge margin on this product, so even with the rebate they were still making fistfuls of cash.

However, the way that our client had been handling the rebates was by having forms that they would mail to people, and then people would fax them back, and then they would call them if there were problems, and so forth. All told, signing up for rebates and receiving them took about 6 weeks. 6 WEEKS! When a competitor to the product came out with an online way of doing it, they quickly lost market share. The competitor had basically a three screen long HTML form, which looked pretty terrible, in my opinion. The error messages in particular were pretty unhelpful. Plus, our client wanted to manage a whole lot of other stuff, basically automating a lot of the drudgery they were doing with manual paper pushing. This was an extremely high-visibility project, as this was the first time they outsourced something like this, and everyone from the VP level down was involved.

When we were all done, people could come in and sign up within FIVE MINUTES typically. We calculated this out one day, and compared to 6 months, it was like 300 million percent better or something.

So anyway, they were losing their market share, so they wanted a great solution as soon as possible. There was significant requirements churn in an effort to get the minimum subset of functionality needed to launch. The scope and time frame kept changing.

We bid on a database-backed web app, using pretty unstandard technologies (at least for us):

IBM's WebSphere 6.0 for the web application server (their IT wanted to use this)

They didn't even support the latest version, which was a real pain point for awhile.

Using WebSphere necessitated developing on a Java compatible framework of some sort

We were scared of using Java this due to the aforementioned time constraints

We had been experimenting with using Ruby on Rails for some internal apps and pet projects, so we decided to try to meet all of the requirements with Rails running on JRuby (was not 100% compatible at the start of the project.)

SQL Server for the database (their IT wanted to use this)

Wanted to support IE 6 because that's what all of their people and probably a good common denominator for pharmacies in the field. Typically the pharmacies were connecting with 56k modems, so we had to assume that they were not running the latest hardware and software.

Developing with IE6, IE7 and Firefox in mind was a pain at times for developing, and slowed our testing down.

Hence, we were running Rails on JRuby on Websphere and connecting to SQL Server. I think we were the only people in the world trying a configuration like this. Keep this in mind for later.

Our development process went something like this:

Lean approach using a Kanban board. This is outside of the scope of this talk.

We used Team Foundation for version control and task management. This was pretty helpful for keeping the work in progress to a minimum and to ensure that things were getting tested and reviewed properly.

When any code is checked in, run all of the unit and functional tests on an integration server

Used CruiseControl as our continuous integration tool

This was very helpful for pointing out when you checked something in that broke the tests, especially if you forgot to check something in, etc.

Finally, we did integration tests with Selenium, but I didn't really mess with that at all, so can't talk about it.

OK, so that's all I'm going to say about the project itself. The take home message was that the project used a pretty complicated structure using unproven combinations of technologies and we were under a fierce deadline with high visibility. Sweet.

So a core methodology that we adopted for success for this project was to use TDD. See, I told you we would get back to that! TDD is an agile process. TDD essentially means that every line of production code that we write has a test written for it before the code is written. Think about that for a second. It was a very major departure from what I was used to, and you might be in the same boat right now. But I have experience benefits of TDD that outweigh the changes that you will have to make to your thinking. TDD can be broken down into a few major steps:

write a test that you believe will fail

run it to ensure that it fails

write the simplest code possible to make the test pass, using interpreter messages and your brain to figure out the next step

run the test suite to see that all tests pass

refactor to clarify and simplify

run the test suite after every refactoring

lather, rinse, repeat

So that is the process in a nutshell.

First off, we write a test that we think should fail. This failure represents the new functionality that we want to add. This step is the hardest one mentally, because it forces you to think about the interface that your code will have with the world. Essentially, and Stephen Covey would love this, you need to begin with the end in mind. You have to think, "what automated test could I perform, so that if it passed, I would believe that the code has functionality that I expect."

I only have about two years of professional experience, and there were times when I didn't unit test at all, and there were times when I put unit tests in there because someone else asked me to after the fact. But I can tell you that thinking about the way you would test the code before you even write it changes the way that you will write it, and not only improves your test coverage in terms of scope and depth,

but also INCREASES the quality of the code produced. You might have one application produced with TDD and one produced with unit tests afterward that have the same functionality. However, I argue that the TDD-produced code and tests will be cleaner and more understandable, produced in less time overall due to higher quality, and more maintainable with less regressions.

Writing a module that stands alone that no one else maintains and that is bug-free is difficult. But when you have dense code that does not codify the assumptions it makes that multiple people are working on at the same time, it makes it impossible to make changes reliably. Have you ever developed on a system and were hesitant to change something because you didn't know what the code did or how it interacted with the rest of the code base? Imagine having the peace of mind that if you changed anything significant functionality-wise, you would immediately know about it. You're not afraid to move code around to make it more understandable because you have a safety net with every refactoring that you do.

If you think about it, every single piece of code that we are checking in has a test written for it. This makes working in a team environment significantly more productive. You immediately know if your changes are what are causing problems, or if it's someone else. This allows you to focus on what you are currently working on, and keep the broken windows to a minimum. The integration server is key. When you check something in, a separate environment is generated that simulates the production environment, and all of the code that is checked in is run against the unit and functional tests. If any tests fail, you are alerted to this fact, and everyone yells at you for breaking the build. This makes sure that you can quickly check something in that you forgot to check in, or fix the test that broke when you changed that function that shouldn't have messed anything up. Instead of taking days for that problem to get back to you, it happens in a matter of minutes. When you get the latest version of code, you know that it is extremely high quality.

Tests developed in a TDD manner actually serve as documentation of assumptions for the code. If you want to know what a method does, you can look at the code. You can also look at the tests to see what assumptions the author has made and how well it is tested. If all of the tests pass and you think they make sense, you might not even need to review the code itself.

Using TDD keeps your cycles short. It's never two hours before you get feedback, it's more like ten seconds. This is very powerful for your morale, as you get a feeling of fixing a broken test, then hungrily writing a new one and fixing that. I will add at this point that a unit test is not a unit test if it takes more than a second to run. You cannot overlook this principle. I think everyone in our group needed to upgrade computers throughout the six month project to ensure that all the tests would run in an acceptable time frame. But maybe that's because we didn't do things quite right. To this end, it is best for test simplicity as well as running speed to have only one assertion per unit test.

OK, so we have a well-written but failing test. So why do we run the test even though we think it will fail? Well, one reason is that someone else might have already added the functionality that we are planning on adding, but we just weren't aware of it. That happens on larger projects with several team members. Sometimes we have a duplicate test name, and Ruby only recognizes one of them in a particular namespace, causing the new test to never run at all. The other reason is that there might be a bug in the test that we wrote, causing it to pass. All of these would be bad, because the success case for the process is that all of the tests are passing. If the new test you just wrote passes, it is either superfluous, reinforces or clarifies some assumptions that you have about the code and so no code needs to be written, or it has a problem of some sort. So while it's a simple idea, and you might be tempted to skip it, it makes your job a lot easier.

OK, so now you have ensured that the test successfully fails. When implementing the code to make the test pass, you strive to have the minimum code possible to get the test to pass. If this means hardcoding in values at first, you should do that. If it means using an array instead of a more complicated structure, use the array first. Obviously you need to use your best judgment, but the reason for this is that you might not need all of the complexity that you think that you will need, and will over-engineer the code. When we get to the refactoring phase, we will make the code look prettier or have reductions in duplication. You can also increase the complexity when new failing tests warrant it. To this end, you cannot add more functionality than is currently being tested. Just get the current tests passing, and then add more tests to cover the functionality that you want to add. This keeps you from getting developer-induced scope creep. You will be very conscious when you are adding things inadvertently.

Next, you run the entire test suite to make sure that your changes didn't inadvertently break something in another part of the application. Then, when everything works, you can safely refactor the code because everything that matters should theoretically be under test. If it really matters and you change something and no tests were broken, then there should have been a test written for it. Make sure that you run the relevant tests after every refactoring to ensure that your refactoring was successful. This helps to keep your change sizes small. If something breaks, you know that it was the last three or four lines that you changed, so it's easier to see what changed. You can do `rake test:recent` to trim down the tests run, but the whole suite should be run before you go on, and especially after getting latest before checking in again.

And then you start at the beginning.

So now for some personal takes or lessons learned.

One important thing that I realized while working with TDD was that it was more important to review the tests to see if some were missing than to review the code itself. You were looking more at the logic than the code itself and possible ways that the code could have been broken. It is pretty pointless to do anything with unit tests and not review the unit tests. You might as well not be doing them.

Rails makes this whole process easy with great unit and functional testing built right in. To clarify, unit tests typically test your models and library functions, while functional tests test your controllers and views. It's nice that your tests are broken down along these lines, it makes things a lot simpler. As a side note, I am of the opinion that Rails' opinionated structure lends itself well to picking up new team members that already know Rails. About 80% of the way through the project we added a new member at little cost to the overall project. He pretty much already knew our conventions and where to look for things because we used the standard configuration. It was pretty important to have that happen to get done on time.

One thing that kind of bit us was the heavy use of test fixtures in code. Rails has this great mechanism for quickly using fixtures for your models. However, the fixtures were typically very brittle, and caused problems to pop up in all sorts of tests. You could change one value and expect ten tests to fail. It was extremely annoying because of all of the dependencies. There were some crazy setups, like load user "Abraham Lincoln" and then have him try to login to the admin section and see that it fails. It was hard to follow the logic of tests like these, since you have no idea why the hell Abraham Lincoln is a test fixture, or what he stands for. We also had some testing helpers that set things up in a similar manner, and they too were very brittle. Towards the end of the project, we started setting things up in a more independent fashion, and that seemed to help. Tests are a large part of the work you are doing, so

don't underestimate the problems of doing them poorly. Obviously we learned as we went.

TDD was helpful when I did pair programming. I think that I only really did this once in the project, because we just didn't run the rest of the project this way. Someone else and I were working on something very similar, so we decided to not step on each others' toes and just do it together so we understood everything. Constantly knowing what you are working on really helps you to stay focused. The person who was driving was typing furiously, and the person sitting back was considering the current test and perhaps thinking ahead to the next test. It was very good for being disciplined, at least with the partner that I had. He was very meticulous with writing tests. I asked him about this later, and he said that he could write code without tests, but it just kind of felt dirty now.

The feeling of security that TDD gave us cannot be understated. We had the clients continually looking at code on a staging server, some of which was not yet fully tested due to the way we handled pushing to the staging server. While it had not gone completely through test (which had integration tests with Selenium), it was still very high in quality, and allowed us to be much more responsive to the clients and show them our progress in a better manner.

What's more, as I mentioned earlier, we used several different technologies of varying degrees of openness. On the one hand, we had closed but mostly stable technologies, and then literally bleeding edge products. I mean the bleeding part. We had to submit various bugs and patches to JRuby to ensure that our system would be working correctly. Our tech lead Matt updated Rails about five times, jumping over the 2.0 marker, and updated JRuby itself through various release candidates and then final versions. Without a solid set of unit, functional, and integration tests, there would have been no way that we could have reliably did this. Every time Matt updated something like this, he found about ten to fifteen tests that broke in very subtle ways. Consider what happens when there is a bug in JRuby date handling, or what happens when the SQL Server driver for Rails changed so that one of our old assumptions are no longer valid but it has some nice fixes that we want to apply. I can't even fathom what it would have been like not having that in place.

OK, I'm almost done.

I'm reading a book called "Working Effectively With Legacy Code." It sounds like an old cruffy book, but it was published in 2005. It has a bunch of strategies for working with legacy code that nobody wants to work with. It has funny chapters like "I have a monster method and I need to make changes to it!" or "I don't understand this code but desperately need to make changes to it!" This book applies to my presentation because of the definition Michael Feathers gives in his book. He says:

"In the industry, legacy code is often used as a slang term for difficult-to-change code that we don't understand. But over years of working with teams, helping them get past serious code problems, I've arrived at a different definition.

"To me, legacy code is simply code without tests. I've gotten some grief for this definition. What do tests have to do with whether code is bad? To me, the answer is straightforward, and it is a point that I elaborate throughout the book:

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."

Most of the book focuses on ways to break dependencies between classes and structures to be able to slowly but surely use the principles of good object oriented design to put the code base on a solid foundation of unit tests. I can tell you that I've worked with a system that was a ten years old and 100k of Perl code that had about fifty tests. It's hard to know if you've fixed something without introducing a new bug. There were pieces that we could clearly see were copy and pasted, but the risk of changing them didn't outweigh the cost of introducing bigger problems. Simple changes took inordinate amounts of time. There were users that did not want to be disappointed.

This naturally begs something for you to think about: Are you currently creating a legacy project?

Hopefully this discussion gave you some food for thought, and that you take action to prevent creating yet another legacy project.