Embracing Nihilism as a Software Development Philosophy and the Birth of the Big Book of Dead Code.

Ryan Bergman
Des Moines Iowa, USA
ryan.bergman@gmail.com
Twitter: @ryber

Introduction

Agile software practitioners often brag about how quickly they can change course with a project as the requirements evolve and change. What happens to the things we no longer need? Are projects removing functionality no longer needed by the project? Are we truly willing to abandon or even (gasp! delete) "dead" code. Do we know dead code when we see it? Even the most agile developer has left unnecessary code because it "might" be used in the future. Even the most agile product manager has left a seemingly useless feature because "someone" might want it.

This paper will examine a case study of a project that was filled with dead code and how a team turned it around. This team was able to drastically reduce code base size, build/test time, application runtime and increase stability by doing nothing more than deleting code nobody needed. We will look at how the team overcame personal egos and pride of ownership to embrace ideas like YAGNI (ya ain't gonna need it) and minimal marketable feature. Part of this great transformation was the production of a "Big Book of Dead Code", that visualized waste and bloat for everyone involved in the project. We will look at how the book came about, and how other teams can create their own.

Background

Our story involves a small company involved in software dealing with "human capital management". The company employed around 250 full time employees including anywhere from 40-50 software developers, designers and testers. While the company's client base included some large corporations and Government agencies it's bread and butter was small to medium sized companies (300-15,000 employees). The company specialized in hosted software delivered on its own servers (never installing behind the firewall solutions).

Setting the Stage: The waterfall begins and the sky is the limit

The project began much in the same way as many software development horror stories. The old version of the project was written in a "washed out" technology. Its business logic was spread across the application and the database. It had bugs, it was slow, it was unstable. It needed to be re-written from scratch! The developers had been banging this drum for some time. The business was having problems scaling. While the application was a web based app delivered by the company on its own servers (never behind the firewall), it could only serve one client per instance. This meant that in order to scale, each client, no matter how small, needed a dedicated app instance complete with IP's, db schemas, etc. A small team of 2-3 developers, designers and sysadmins would take around a week to get each client set up. This included server space, designing a custom theme, and manual configuration of application roles and permissions. If the client wanted custom rules built into the application it would require even more time. Some large clients would take months of work by much larger teams of developers. These engagements had a habit of swallowing up the entire IT department.

The business realized they needed to get out of the one-off game. Life with the current app was simply too expensive. If the business wanted to grow they needed to move to a SAAS (Software as a Service) model (think SalesForce or Google Apps for companies). This would require a single application where new clients could be set up quickly by a salesperson. Everything that needed to be configured (themes, permission, rules) should be able to be done

978-0-7695-4804-3/12 $26.00 © 2012 IEEE
DOI 10.1109/Agile.2012.14

by the client themselves without technical input. The most radical change was in the implementation of those large clients. All work would now be done in the single codebase. Those client who had asked for customization in the past would now need to ask those features to be placed on the single backlog. Features that didn't make sense for all customers to have access to would not be done at all.

The new project would be developed in classic waterfall. The company needed its current developers to keep maintaining the legacy app so new contractors were brought in to develop the application. At first only a core group of 3-4 was formed. Their job was to help produce the architectural specs for the app. Various product managers from the company were given areas of the app to own and produce requirements for with the contractors. All requirements needed to be done before development began so the contractors could be as efficient as possible.

The resulting spec was a behemoth. Each of the product managers gold plated their specs to the hilt. They felt this was their one shot at getting everything they wanted in. There was also little priority between the managers so user stories just got in line regardless of how much value it brought to the product. Finally the contractors had no investment in saying "no" to any request. For them it was just more work and of course all of it could be done. The most painful example of this was a technical decision about platform. The business knew it wanted the application written either in .Net or Java due to the requirement in many of the RFP's they answered (this, despite the fact that the company delivered its products as a ASP/SAAS, so what does it really matter anyway?). Exactly which platform to use became a large sticking point with various camps forming around each platform. Finally in order to appease everyone the contractors proposed that the company could have its cake and eat it too. The application would be written in C#, but written in such a way that it could be ported to Java "within a month". This meant using frameworks that existed in both languages when possible (N/Hibernate) and writing completely new frameworks from scratch when it was not (page rendering, URL routing, MVC, reporting, security and authentication … pretty much everything else). The application would even have a custom build and packaging tool for managing CVS and the .Net solutions and projects so that developers would not be "dependent" on MS Visual Studio (it also rendered actually using VS or CVS natively difficult to impossible).

Because of all of the custom frameworks, once the contractors started working it was almost a year before anyone saw anything, and that was basically just a log in and landing screen. Almost no work had been done on anything that actually made the app something the customers would want. Pressure became intense. Large customers demanded to see a product that was going to be an improvement over the existing app and would solve their problems. The developers began working long hours to get features ready for conferences and demos. It became known around the office as "demo driven development" and all it needed to do was look like it was working.

Because the software was waterfall and because the "core" team had theoretically provided the frameworks, feature work was outsourced another level to eastern europe and elsewhere. As deadlines came and went more developers were added to the mix. At it's peak there were around 30 contracted developers. Most of these were not co-located, and those that were, were in eastern Europe. The primary architects all existed in the western United States. There were huge lags in communications between these groups which resulted in fragmented work and high WIP.

These groups were basically working on all features at once without any good feedback or defined outputs. They had the original specs and would toil for weeks or even months on particular features with little guidance. When they did finally get input it was not unusual for there to be large changes resulting in abandoning parts of what was already written. This dead code was was never removed and if you were lucky it would get a comment saying it was deprecated.

Almost none of the contractors at any level had any kind of background in writing an app in the companies domain. The full time developers back at the company who had years of experience in the same problems were kept at odds length and mostly only brought in to talk about what existing customers expected out of features. This resulted in the contractors making many of the same technical mistakes the legacy developers had made over the years in creating and maintaining the legacy app.

The Home Stretch

Development didn't get easier. Unit testing was given lip service at first but was soon abandoned

because it was slow and brittle. The code was tightly coupled to Nhibernate and almost all tests required a clean database. Tests were also highly order dependent and the final straw fell when nUnit was updated to a new version that changed the order of the tests and broke almost all of them. QA testing was nonexistent.

Eventually the time came for the contractors to transition the application to the full time developers. The application by this time was quite large and complex. After QA finally got their hands on the app it was discovered that almost every part of the system had numerous bugs (many of them serious). The lack of unit tests, coupled with the challenge of training teams of ColdFusion developers to a non standard C# application resulted in even more bugs and issues being introduced. The teams limped along for 6 months doing nothing but fixing bugs until the application was in a “mostly seems to work” state. It was at this point that the first clients were introduced.

As a SAAS application it was decided to try and release monthly. Releasing the app was a confusing mess of perl scripts and manual interactions. Database changes proved very problematic and it was not uncommon for a deployment to take all weekend. Clients were unhappy, the business was unhappy and most of the developers were starting to put their resumes together and openly talk about interviews and who was hiring.

The Switch to Agile

It was at this time that the business made a sea change and brought in all new management. While “scrum” had been practiced for years, XP practices had not, and scrum was typically done a-la waterfall.

From here on out all new code and bug fixes needed to be covered both in unit tests and functional tests via Fitnesse. The entire idea of keeping the application in a state where it could be ported to java was abandoned in favor of being able to actually use modern .Net features like generics and method delegates. In order to give developers the refactoring tools they needed the custom build tool was stripped out and proper visual studio projects were created. CVS was abandoned in favor of more modern VCS (They eventually settled on Git after trying out a few).

It was at this time that the true size of the application came to view. Previously, developers wanting to work in VS could only work on one project at a time. Now they could see just how big the application truly was. Including the new .test projects the solution had over 140 projects. It took almost 10 minutes just to load into a typical developer's VS2008 instance. A build took another 7-8 minutes and unit tests took well over 20 minutes for the few hundred that existed and actually worked.

By this stage most of the original architects were gone. There was nobody left to defend existing design. The development staff felt no personal ownership towards the code. The first problem to tackle was obvious. We had to fix the productivity problem. We had a giant room full of highly paid software developers who were spending the majority of their time watching Visual Studio. A few developers splintered off with the mission to dramatically cut the code/build/test lifecycle.

The Big Book Of Dead Code

The first things to die was the code that was truly dead. Code for modules that were started and abandoned at some time. Often these modules hooked into real production code but were configured “off” in some way. Some of them didn’t even do that. After a few weeks of deleting this code the team had reduced the solution to about 100 projects. This, coupled with some optimizations to test startup and tear down gained 10 minutes total in a developer build/test cycle. VS2008 loading was also much faster.

All together some of the things removed included:

- Two entire handwritten reporting engines.
- A Lucene based search engine wired into NHibernate
- A dynamic permissions / rules system that was supposed to let people script complex rules in XML (they never did which is good because it didn’t actually work).
- A central scheduling graph system for calculating time conflicts that crashed the system every 15 minutes.
- Multiple implementations of common objects with only one of them actually being used (Group, GroupNew, Groups, Group2)

The sheer amount of code that was dead was frightening and answered some of the questions of why initial development took so long. An idea was hatched for a good way to visualize this waste to developers and management. A small program was written which analyzed diffs and pulled out removed files. The code was reformatted to remove all whitespaces and turned into a solid block of 120 characters and printed out double sided. Over time the application was reduced by over 1200 classes or almost 25% of the original codebase. The resulting book was well over 500 pages. The purge continued on with most developers adding to the book in a kind of competition to see who could add the most. After about a year the project had around 40 projects total. It would load into VS in under a minute. A full build/test cycle including over 14,000 unit tests and 1000 fitnesse tests took under 12 minutes. It was easily one the most impressive technical feats I've ever seen.

There was no reason that application should have lived. Almost anything that could be wrong with an application was with this one. The team had rallied around its awfulness. Making it better was the side project of many of the developers and they would put in extra time on night and weekends just to kill off or refactor their biggest pet peeves. It was all celebrated all the time.

Figure 1: Example Ruby/Git dead code generator:

```
Repo.new(".").commit_diff(commitHash).each do
    |dif| dif.diff.each_line do |line|
        if(line[0,1] == '-')
            result << line[1,line.length].gsub(' ','')
                                         .gsub(' ','')
                                         .gsub(/\n/,'')
                                         .gsub(/\r/,'')
        end
    end
end

puts result.scan(/.{1,120}/).join("\n")
```

The Book's Impact on Behavior

The existence of the book had different impact on the different areas of the business. Most of it positive and revolving around greater transparency and communication in order to reduce the possibility of another such book in the future.

Management

The introduction of the book to management was a sobering experience. The book represented an enormous amount of time and money. They had been quite concerned with the experience of the new application being made and knew they never wanted to repeat it. They felt the development was too slow and too much of a black hole. What they learned was that there was quite a lot of code being written but a lot of it was never seeing the light of day. This began many conversations about the roadmap. How would they ensure that developers were working on the right things, how would they ensure that the clients were seeing the value and that the features were driving sales?

The business had already begun to fully embrace agile and many changes had already kicked in. The book helped to inform some of these changes including:

1. A strong aversion to using outside contracting groups to develop software core to the business.

2. A distrust of "technical" stories. It became quite difficult for developers to convince management that there were technical stories that "had" to be done. The book had been filled with technical stories that were not bringing the business direct value.

3. A much stronger interest in the day to day process of software development. One of the biggest issues with the project previously was the way developers would disappear for weeks on a feature without any input (or output). Teams were organized into scrum teams which operated in two week sprints. Demo's were given to management at the end of each sprint and teams needed to commit to delivering working software. Finally the backlog was unified into a single priority list. Product managers needed to make the case that their features were the most important.

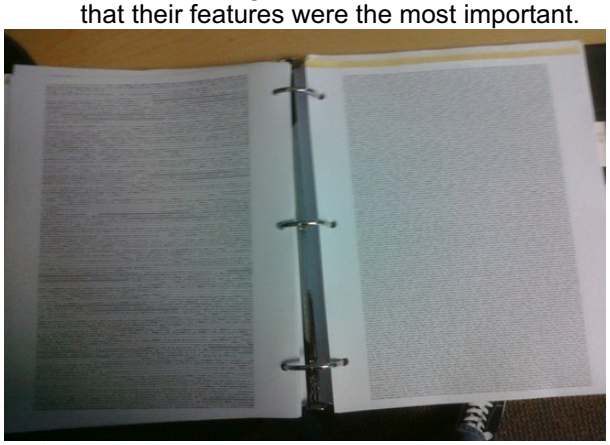

Fig 2.a

Fig 2.b

Developers

To say that developers were fascinated with the book would be an understatement. This represented more than just time and money. It was their own time, effort and personality. There were things in the book that represented months of peoples time. Developers often like to think that the code they are writing will be around forever. This flies in the face of the conventional wisdom that most software projects fail. If most projects fail and most code gets thrown away then why are developers gold-plating or over-engineering code in order to make it “flexible” for the future? The hard facts of the code mortality rate began to challenge this view. Flexibility and robustness were no longer in vogue. The product was still not entirely stable or fast and the backlog of work was quite large. The idea of spending extra time to make something flexible (particularly something that might end up in the book) became unattractive to the development staff.

The fact that this state was achieved through a nihilistic attitude about the project was a concern for some. I would contend however that the nihilism was the projects saving grace. Most of the developers now on the project had only a peripheral connection to the original development. They had the advantage of understanding some of the reasoning behind why something was developed the way it was without having any emotional attachment to it. Additionally most of the contractors responsible for the architecture were gone so there were no “code-baby-mammas” to defend the way things were. Developers were free to delete and change code at will. This attitude eventually extended to some of the developers own new code. Developers saw what happens when you put too much personal stake into code. They no longer wanted to invest the emotional attachment needed to defend it.

This is not the same as not caring about the project or the company or being entirely cynical. Developers simply came to see code as a product that needed to do something and nothing more. It was not an extension of yourself, it was not a platform on which other peoples code needed to be built. Developers began to talk quite a lot about things like yagni and simple design. They wanted classes that were very loosely coupled and had full test coverage. If anything the developers became even more interested in writing good, clean code. There was so much work to be done that developers wanted to make sure that the code they wrote was not something they would have to return to as a defect or to spend a lot of time explaining to another developer. In addition to this developers began practicing pair programming. This helped contribute to the idea of group ownership and an inability to spend lots of time gold plating solutions.

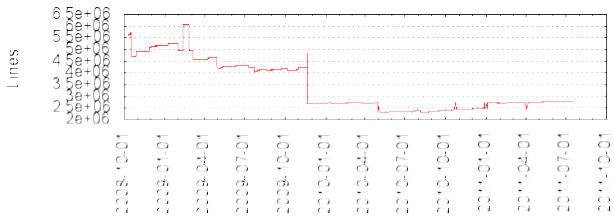

Fig 3.a: Lines of Code over time. Note that once the code base stabilized the total number of lines did not go up significantly. This entire time the team was adding features. This represents the code history once it was moved to svn (and later git). Earlier history was lost in the transition.

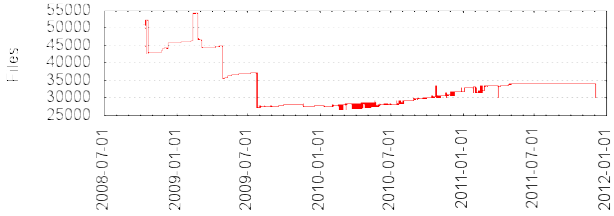

Fig 3.b Number of files over time.

Summary

In summary the book of dead code provided an invaluable visualization tool for developers and the business. So many of our software projects fail and unlike a building or highway that never gets completed we never get to see the debris of our labors. Weeks and months poured into a IDE often just disappear in the wink of an eye. By visualizing the work we do, we can appreciate both the time that went into the project and temporal nature of code. This allows both developers and product owners to better form opinions on how features and code should be prioritized and written.