Extracted from:

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

David Scott Bernstein
Foreword by Ward Cunningham

edited by Jacquelyn Carter

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

David Scott Bernstein

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Printed in the United States of America.
ISBN-13: 978-1-68050-079-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—June 17, 2015

CHAPTER 13

Practice 9: Refactor Legacy Code

Refactoring is restructuring or repackaging the internal structure of code without changing its external behavior.

Imagine you're where I was a few years ago, saying to your manager that you want the whole team to spend two weeks, a full iteration, refactoring code. My manager said to me, "Good. What new features are you going to give me?"

And I had to say, "Wait a minute. I'm talking about *refactoring*. Refactoring is changing the internal structure but not changing the behavior. I'm giving you *no* new features."

He looked at me and asked, "Why do you want to do this?"

What should I say?

Software developers are faced with this situation too often. Sometimes we don't quite know what to say, because we don't speak management's language. We speak developers' language.

I shouldn't tell my manager I want to refactor code because it's cool, because it feels good, because I want to learn Clojure or some other piece of technology… Those are all unacceptable answers to management. I have to specify the reasons for refactoring in a way that makes sense to the business—and it does.

Developers know this, but need to use the right vocabulary to express it—the vocabulary of business, which comes down to *value* and *risk*.

How can we create more value while at the same time reduce risk?

Software by its very nature is high risk and likely to change. Refactoring drops the cost of four things:

- comprehending the code later

- adding unit tests
- accommodating new features
- and doing further refactoring.

Clearly, you want to refactor code when you need to work with it some more to add features or fix bugs—and it makes sense in that case. If you never need to touch code again, then maybe you don't need to refactor.

Refactoring can be a very good way to learn a new system. Embed learning into code by wrapping or renaming a method that isn't intention-revealing with a more intention-revealing name. You can call out missing entities when you refactor and basically atone for any poor code written in the past.

We all want to see progress and meet deadlines, and as a result we sometimes make compromises. Refactoring code cleans up some of the mess we may have made in an effort to get something out the door.

Investment or Debt?

I first told the following story on my blog in April of 2009:[1]

> Some businesses think software is a write-once activity. It is true that once written software does not change, but the world around us does and static software quickly becomes out of date.
>
> Code rot is real. Even if a system was well written at one time we often cannot anticipate how a system will need to be changed in the future. And that's the good news! If software does not need to be changed then it probably isn't being used. We'd like the software we build to be used and for software to continue to provide value it must be easy to change.
>
> You can build a house out of cardboard and on a nice summer day it will hold up well but in the first rainstorm it'll collapse. Builders have a whole set of standards and practices they follow religiously to make their buildings robust. We as software developers must do the same thing.
>
> I have a client on the East Coast who is one of the leading financial companies on the planet. They are particularly challenged by their huge amount of legacy code. Most of it was built by contractors or their top developers who were pulled onto the next project as soon as the first version was completed. Junior developers now maintain those systems, some who do not understand the original design so they hack away at it to make changes. Eventually they ended up with, well, a mess.
>
> At a meeting with several of their senior managers and developers I said, "So let me get this straight. You became a leader in financial management by going out

1. Bernstein, David Scott. *To Be Agile* (blog). "Is Your Legacy Code Rotting?" http://tobeagile.com/2009/04/27/is-your-legacy-code-rotting

> and finding top fund managers to manage your funds, have them research and acquire the best investments, and then you freeze the portfolio and pull the managers off onto other projects.”
>
> They said I got the first part right but the fund managers stay with the funds and continually adjust their portfolio because the market is constantly changing.
>
> “Oh,” I said, “so you go out and hire top software developers, have them design and build a system and then you pull them off when they’re finished to work on a different project.”
>
> “So are you saying that our software is a critical asset, and like any asset it needs to be maintained?” one of the managers asked.
>
> Bingo.

Would you buy a $90,000 Mercedes then insist on never taking it to the shop because you paid so much for it? No. No matter how expensive and well-made the car, it still needs some maintenance. No one builds a house thinking no matter how long it may stand they’ll never replace the carpeting, buy a new kitchen appliance, or throw on a coat of paint. On the other hand why replace the transmission on that Mercedes three days after pulling it out of the lot just because you’re pretty sure you’re eventually going to have to fix it? But if the day after you pull it out of the lot it won’t go into reverse, how long do you just drive it around trying not to have to back up before you get that transmission looked at?

Some things are good to put off until later. Some things are not good to put off until later. Knowing the difference between the two is absolutely critical.

For technical debt, for the things that accumulate, almost always—and there are definitely exceptions to this, but most of the time—paying off technical debt as soon as possible is the right choice. When technical debt is lying around in the system, and developers are working on that system, they’re bound to have collisions. They’re running into that technical debt and paying the price over and over again. They can’t get the car into reverse so they start to modify their behavior—their driving habits and routes to and from places—in order to never have to put the car in reverse. That one problem is starting to cause more and more problems. The sooner you deal with technical debt, the cheaper it is—just like credit card debt.

Become a “Deadbeat”

Technical debt really is very much like financial debt, the interest can consume you.

I've worked with the financial and credit card industries, and there's a term they don't like to advertise but that they use for people like me. I'm a person who always pays his bills in full, as soon as I get the bill. I never let any balance accrue. Credit card companies call customers like me "deadbeats." They hate us because they don't make any money on us. They love the guy who builds up a balance and keeps making the minimum payment. I know someone who owed a major credit card company $17,000. If he paid only the minimum payment due each month it would have taken him 93 years and $184,000 to pay that back.

Like financial debt, ignoring the problem doesn't make it go away. I want you to become a technical "deadbeat."

Sometimes, we do have to let technical debts lie there for a little bit, either because it's not the right time to fix it because we don't know what to do, or simply because we have no time right at that moment. We end up having to live in that world more often than we'd like. But that's the difference between making the minimum payments on a credit card for a few months to get through a rough patch then paying it all down plus a few dollars in interest, and just pretending all's well until sometime early in the twenty-second century.

We're not trying to create perfect code. I just can't stress this enough. Perfection is something that no one will achieve—and software developers don't *want* to achieve. We have to always be aware that there are trade-offs. Do I ship with problems in code sometimes? Yes. I have to. If I didn't, I would go out of business.

When Code Needs to Change

Even the worst-written legacy code can continue to provide value if we simply leave it alone—as long as it doesn't need to change.

This kind of judgment call should be made on a case by case basis. Mission critical software requires different standards than video games. The good news about software is that, unlike physical machines, bits don't wear out. But what about the legacy code that *does* need to be changed or extended?

When software is actually used, people find better ways to use it and that leads to change requests. If you want to support your users in finding more value in the software you've created for them, support them as their needs change by finding ways to safely improve your code.

Now that we know some of the characteristics of good code, use the discipline of refactoring to safely and incrementally change your code to be more maintainable and extendible. Take poorly designed legacy code and apply safe refactoring that gives you the ability to inject mocks and make software testable so you can retrofit unit tests into the code. This safety net will allow you to perform more complex refactoring to safely accommodate new features.

Cleaning up legacy code in this way, making incremental changes, adding tests, and *then* adding new features, allows you to work with legacy code without the fear of introducing new bugs. When you have the right unit test coverage you can safely drive new development and refactoring from the green bar. This is a much safer and cheaper way to drive changes to software.

As an industry, we have a lot of code—legacy code—that isn't working as well as we need it to work, and that's all but impossible to maintain, let alone extend. We have mountains of legacy code out there. But what can we do about it? What *should* we do?

For the most part, nothing.

As an industry, we need to see legacy code not as *time bombs*, but as *land mines*. If code is working, and it doesn't need to be changed or upgraded, then leave it alone. That goes for the vast majority of all the legacy code in the world. Like they say: "If it ain't broke, don't fix it."

As soon as we start mucking around in legacy code we're bound to cause problems. If code is working the way you want it to, keep using it. This is true for most of the existing software in the world. Generally speaking, the code you want to refactor is the code that needs to be changed.

If there are bugs or if there are features that need to be added or modified, then it makes sense to go in and make changes to existing code. Changing code is risky and expensive, so be prudent about doing it. But when it does make sense to go out and change code, do so using a methodology that allows it to be done safely. It turns out that the techniques for doing this are the same techniques we've been talking about for writing good quality new code.

You can refactor existing code just like you refactor new code.

Retrofit Tests Into Existing Code

While this can sometimes be a lot more challenging than writing code test-first, because it can require changing existing code so that it's written to be more testable, the overall effect is to improve the maintainability and drop the cost of change to existing code.

Some code is heavily used. Some code is likely to need to be changed. Change requests are good. It means that someone cares and wants to see some of our code improved.

When this happens you want to be able to respond and provide new features for existing software so your existing customers can get more benefit from using it. Of course, there's a lot of code out there that no one cares about anymore. That code can quietly rot away, but the bits that are used, the ones customers depend on, that are likely to change—that's the code you want to target for refactoring.

Refactor Bad Code to Learn Good Habits

Refactoring is a discipline that not all developers are currently aware of, but refactoring can also be a craft that can help build good development habits and demonstrate how to build more maintainable code. These are the skills that will always be in high demand for software developers.

Refactoring legacy code sounds boring but it's actually quite exciting and challenging. It gets far easier with a little practice and when you get good at refactoring, something very interesting happens: You stop writing bad code and following poor development practices and begin *pre-factoring* (see *Prefactoring [Pug05]*), writing cleaner code to start with. Refactoring code is one of the fastest ways I know to learn what *not* to do when writing software and what *to* do instead.

Postpone the Inevitable

Our goal as software developers is to create value by building valuable software. This means the software we develop has to create value now and also continue to produce value into the future.

In order for software to continue to produce value into the future, the cost of ownership must drop so it's cost-effective to improve and extend software. Make the supportability and maintainability of software a priority and the cost of ownership for software will drop.

But at some point, like all things, software must die. I'm surprised at how long some of the software I've written has lived. I wrote code when I was a kid that I wasn't proud of then but that has lived on in some form to this day.

Software tends to either die on the vine or outlive all of our expectations, and you can never accurately predict when a piece of code is written whether it will be one kind or the other. But we all want the time our software has to be of high value. The return on investment should be as high as we can make it, while at the same time drop the cost of ownership.