Editor: George Fairbanks
gf@georgefairbanks.com

Ignore, Refactor, or Rewrite?

George Fairbanks

IMAGINE THAT YOU have some code written, but it has problems. The problems are small enough that you could imagine rewriting the code completely, and you must choose what do. You could do nothing (ignore it), make incremental changes (refactor it), or write new code from scratch (rewrite it). How do you choose? What factors do you consider?

There's already a lot of guidance. In fact, the very existence of *refactoring* on the list of choices is special because the idea of refactoring code wasn't well formed until the 1990s. When you refactor code, you make changes that improve its structure but do not change its visible behavior, and our tools are increasingly good at supporting refactoring, helping us make sweeping changes safely.

Most of the guidance applies to smaller chunks of code and decisions implemented in hours or days, not weeks or months. I've long wished that there were a great book on architecture-scale refactoring with distilled wisdom and case studies of successes and failures. This article only touches on that but it covers some topics that augment the good advice you will find in Martin Fowler's *Refactoring* book[1] and Michael Feathers' *Working Effectively with Legacy Code*.[2] For a good case study of architecture evolution and decision making, I suggest Tony Tsakiris's article[3] in which he describes how Ford looked at its vehicle control interfaces across many car brands and how it chose to embrace, tolerate, or deprecate each of them.

Digital Object Identifier 10.1109/MS.2018.2880662
Date of publication: 22 February 2019

Keep Yourself Honest

Engineers are eager to fix problems because the act of fixing things gives them joy. I find that I must tamp down my enthusiasm so that I can make a good decision, not just throw myself into fixing all of the code. Here's how I keep myself honest when choosing between ignore, refactor, and rewrite.

First, I make a three-part prediction. The prediction will have a lot of guesswork, but the decision-making process helps me keep the over-eager parts of my brain in check.

1) How long will it take to refactor or rewrite? When the project is large, the opportunity cost of refactoring or rewriting is similarly large. Spending a few months refactoring means features are delayed.
2) How long will it take to break even? After my investment, I expect to be building features faster and better—when exactly will that be?
3) What will be different afterward? Architecture-scale refactorings may not deliver features directly but instead improve qualities of the system, e.g., make it easier to onboard new people, diagnose runtime failures, or avoid dependencies that were causing trouble.

Second, I decide whether I can afford the work. The project might be under time or financial pressures that make it impossible to do anything but ignore the problems. But even if I can afford it, maybe it's better to do it next year than now. During the rework, the design will get worse before it gets better. With a big refactoring, there is going to be a long transition period when there is a mixture of designs in active use. Am I okay with that state? The transitional chaos can lead a team off course, especially without great communication about the plan and the transition period.

Third, I try to be as specific as possible about what I don't like about the existing code because different diseases need different medicine. Is it just messy and slowing our feature velocity? Or is it impeding an architectural quality like scale or latency? Both of those are technology issues, but it could also be a problem with understanding the domain—more on that later.

Finally, what's the end game? If I refactor, then the code will be integrated the whole time, but if I rewrite, then I cannot assume that integration will be easy. The bigger the rewrite is and the longer it takes, the harder it will be to integrate. To some extent you can predict how hard the integration will be by digging into requirements and testing coverage of the existing code. If the requirements are not known and the test coverage is poor, you will need to schedule time to rediscover them; otherwise, integration will be trial and error as you asymptotically approach the old system's features.

The Case for Rewriting

It's less risky to refactor code in place than to rewrite it and hope you can integrate. So why should you even consider rewriting? A rewrite is a big jump and a risky one, but it can also be cheaper because making many small moves takes longer than making just one big one. Here are some conditions under which I'd consider rewriting:

- The code is so tangled that it's easier to rewrite it without baggage and to suffer through a tougher integration than to untangle it bit by bit.
- The domain model that's expressed in the code is terribly wrong or outdated. It can be easier to rewrite and integrate because, during a long refactoring, both the old and new models are active in the code (e.g., the old account model and the new account model that have different invariants and concepts), making it quite tricky.
- When you move to a different language, framework, or architectural style, you must commit fully or not at all, which means a rewrite. These are architecture changes.

For all of these cases, it's easier if you can divide up your system because then you can choose between ignoring, refactoring, and rewriting each piece. The microservices architectural style has the goal of keeping that option open. It's safer if you can divide your system and rewrite it piece by piece, but that's still a series of rewrites.

So, there are times when rewriting is the least bad option, but when you decide to rewrite a big chunk of your system, expect a tough road. Your architecture has many dimensions and many stakeholders. It's hard to realize how the old system is satisfying all of them, and, inevitably, some forgotten quality will be worse with the new system, and some forgotten stakeholder will complain loudly about your new design.

Software engineers are not alone in having a hard time reasoning through all of the implications of a significant change. It happens in buildings, too. Tom Wolfe[4] described such a failure in the Bauhaus style of buildings: "No eaves; so that very quickly one of the hallmarks of compound work, never referred to in the manifestos, became the permanently streaked and stained white or beige stucco exterior wall." The architects didn't like the look of the roof overhanging the wall and found out the hard way why old buildings have that design.

Another example that hits closer to home and helps teach me modesty is hearing about a recent project that used an append-only datastore of events. The design succeeded in many dimensions and made several difficult problems quite tractable because they could be expressed as pure transformations of the immutable data. However, eventually, some private information about a customer was saved into the datastore—info that regulations said could not be there—but there was no easy way to remove it because, by design, the data could not be changed after it was written, and every action the system had taken after was logically based on that event being there. It's always the dimension of the problem that you forget about that will haunt you.

So be warned! You may be forced into rewriting instead of refactoring, so do your homework and hope for the best.

Cultural Change

Let's return to that first option, *ignore*. As we think about what could go wrong when we refactor or rewrite, ignoring problems looks tempting. The adages "Let sleeping dogs lie" and "If it ain't broke, don't fix it" are our ancestors warning us. But that advice, however well intentioned, is dangerous on software projects. If your father-in-law advises you to be modest in your weekend projects on the house for fear of breaking things, that's good advice, because your house just isn't that complicated. That same attitude applied to a growing software project allows complexity to rise steadily, and it will overwhelm you.

The broken windows theory of software says that a single broken window in a neighborhood is a signal to all of its residents that it's not worth taking care of things, which leads to more negligence. In contrast, when everything is neat and tidy, people get the message, and no one wants to be the first one to leave a broken window unfixed.

When you lead a team, you want to create a garden where good ideas can flourish. Ignoring code problems endangers your garden by shifting the culture toward poisonous practices. If the code is already ugly, why not add another nested IF statement? If the abstractions are already broken, what's the harm of one more hack? If the test coverage is already low, how can I possibly turn that around?

Code runs on passionless machines, but it is written by warm-blooded people. The team's culture and attitude aren't checked into version control, but they are real all the same. Every project acts as a teaching laboratory where we learn the lessons we will apply on the next project. In that laboratory, you want to reward good designs rather than creative use of duct tape. Some of your team may have only developed code where there are broken windows everywhere, so you may have considerable work ahead of you to show them there's another way.

Unless your code is already squeaky clean, there will be problems you must ignore, at least for now. Be aware that doing so makes it harder to develop the team culture that you want, so you will need to communicate the difficult distinction that ignoring the problem over there doesn't mean the team should write that kind of code over here.

Strengthening Theories

In deciding between ignore, refactor, and rewrite, consider whether the code is comprehensible. It's tempting to ignore code that seems to be working, but let me suggest another way of looking at it that might lead you to refactor or rewrite.

There's a mechanical view of software that sees working code as a machine that is valuable to its owner, much like a coffee maker, light bulb, or photocopier. From that perspective, working code is a tool just like any other machine.

But traditional machines like these don't have the influence that software does because they don't embody ideas the way that software does. If I see an office with machines like a coffee maker, light bulb, and photocopier, it's hard for me to infer anything about what that office does. On the other hand, if I look at the software for that office and I see that it scores the risk of applicants and decides to offer them loans, I'm looking at the core ideas of that business.

When people use nonsoftware tools to get things done, their ideas stay in their heads. Software is different because, as programmers, we find it easiest to reason through programs when the programs express the problem we're trying to solve. From a mechanical perspective, our software would run fine with variables like *x* and *y*, but we instead push our ideas out of our heads and into the code by naming variables *totalSales* and *lastKnownAddress*. That not only helps us communicate with others, it also relieves the burden of remembering what the variable *x* means to us.

Peter Naur[5] calls these ideas a *theory*. When we refactor or rewrite code, it's often because our theory has changed, and we need the software to catch up. The programmers on a team communicate with each other to share this theory as it changes over time, sometimes talking to each other, sometimes writing documents, but always using code. The code isn't a perfect carrier of the theory, but it can be surprisingly good.

However, when code expresses the wrong theory, it will briskly escort its readers down the wrong path. Imagine that I textually rename *totalSales* to *montlySales* and vice versa: at first you won't notice, then you will be actively confused, then you will devote mental effort to swap the meanings in your head each time you encounter them.

Inevitably, some forgotten quality will be worse with the new system, and some forgotten stakeholder will complain loudly about your new design.

Ward Cunningham[6] coined the term *technical debt* to explain to management why he needed to refactor his code that expressed an outdated theory of financial instruments. His team could keep delivering features, he said, but at a slower and slower pace, because they'd misunderstood the domain and would have to work around the code that embodied that domain misunderstanding.

In big and small ways, the decision about what to do with problematic code is something that we decide every day. There's good advice about how to decide when the scope is small, say, when the repair takes a few hours. On a large scale, however, there's less advice, it is harder to decide, and more people must be involved in the decision, such as project, program, and people managers.

Decisions about software architecture are about three things: tradeoffs, tradeoffs, and tradeoffs. Your desire to fix problems in the code trades off against company revenue and commitments to deliver features. We can't simply maximize code health, we must instead contribute to a making a good decision.

This article has laid out my way of thinking about decisions. There are topics that are relevant to good decision making that, in my experience, are rarely discussed, such as the cultural dynamics that affect code quality and how well the code expresses theories about the problem domain and the architecture.

Long-term health of the code depends on the decision makers having the right information and knowing the implications of their decisions. As someone who reads and writes software for a living, you have a special role: you must inform the others about what's happening in the code because what they know about the code comes only from you. If you are able to collaborate with the decision makers and bring the information about tradeoffs happening in the code, then they will avoid the temptation to decide based simply on features and timelines, a short-sighted approach that can lead us to never change the oil in our cars because we simply must get to our appointments.

ABOUT THE AUTHOR

GEORGE FAIRBANKS is a software engineer at Google. Contact him at gf@georgefairbanks.com.

References

1. M. Fowler, *Refactoring*. Reading, MA: Addison-Wesley, 2018.
2. M. Feathers, *Working Effectively With Legacy Code*. Englewood Cliffs, NJ: Prentice Hall, 2004.
3. A. Tsakiris, "Managing software interfaces of on-board automotive controllers," *IEEE Softw.*, vol. 28, no. 1, pp. 73–76, Jan.-Feb. 2011. doi: 10.1109/MS.2011.11.
4. T. Wolfe, *From Bauhaus to Our House*. New York: Farrar, Straus, and Giroux, 1981.
5. P. Naur, "Programming as theory building," *Microprocessing Microprogramming*, vol. 15, no. 5, pp. 253–261, May 1985. doi: 10.1016/0165-6074(85)90032-8.
6. W. Cunningham, "The WyCash portfolio management system," in *OOPSLA '92 Addendum to the Proc. Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, 1992, pp. 29–30.

PRACTITIONERS' DIGEST

(continued from page 129)

3. N. Rios, R. Oliveira Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: First results from a global family of industrial surveys," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM 18)*, 2018.
4. A. Armaly, P. Rodeghero, and C. McMillan, "AudioHighlight: Code skimming for blind programmers," in *Proc. 2018 IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, pp. 206–216.
5. J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *Proc. 2018 IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, pp. 80–91.
6. A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software engineering challenges of deep learning," in *Proc. 2018 44th Euromicro Conf. Software Engineering and Advanced Applications (SEAA)*, pp. 50–59.