Is High Quality Software Worth the Cost?

29 May 2019

Martin Fowler

PROGRAMMING STYLE

PRODUCTIVITY

PROJECT PLANNING

TECHNICAL DEBT

CONTENTS

SIDEBARS

A common debate in software development projects is between spending time on improving the quality of the software versus concentrating on releasing more valuable features. Usually the pressure to deliver functionality dominates the discussion, leading many developers to complain that they don't have time to work on architecture and code quality.

Betteridge's Law of headlines is an adage that says any article with a headline or title that ends in a question mark can be summarized by "no". Those that know me would not doubt my desire to subvert such a law. But this article goes further than that - it subverts the question itself. The question assumes the common trade-off between quality and cost. With this article I'll explain that this trade-off does not apply to software - that high quality software is actually cheaper to produce.

Although most of my writing is aimed at professional software developers, for this article I'm not going to assume any knowledge of the mechanics of software development. My hope is that this is an article that can be valuable to anyone involved with thinking about software efforts, particularly those, such as business leaders, that act as customers of software development teams.

We are used to a trade-off between quality and cost

As I mentioned in the opening, we are all used to a trade-off between quality and cost. When I replace my smart phone, I can choose a more expensive model with faster processor, better screen, and more memory. Or I can give up some of those qualities to pay less money. It's not an absolute rule, sometimes we can get bargains where a high quality good is cheaper. More often we have different values to quality - some people don't really notice how one screen is nicer than another. But the assumption is true most of the time, higher quality usually costs more.

Software quality means many things

If I'm going to talk about quality for software, I need to explain what that is. Here lies the first complication - there are many things that can count as quality for software. I can consider the user-interface: does it easily lead me through the tasks I need to do, making me more efficient and removing frustrations? I can consider its reliability: does it contain defects that cause errors and frustration? Another aspect is its architecture: is the source code divided into clear modules, so that programmers can easily find and understand which bit of the code they need to work on this week?

These three examples of quality are not an exhaustive list, but they are enough to illustrate an important point. If I'm a customer, or user, of the software, I don't appreciate some of the things we'd refer to as quality. A user can tell if the user-interface is good. An executive can tell if the software is making her staff more efficient at their work. Users and customers will notice defects, particularly should they corrupt data or render the system inoperative for a while. But customers and users cannot perceive the architecture of the software.

I thus divide software quality attributes into **external** (such as the UI and defects) and **internal** (architecture). The distinction is that users and customers can see what makes a software product have high external quality, but cannot tell the difference between higher or lower internal quality.

At first glance, internal quality does not matter to customers

Since internal quality isn't something that customers or users can see - does it matter? Let's imagine Rebecca and I write an application to track and predict flight delays. Both our applications do the same essential function, both have equally elegant user interfaces, and both have hardly any defects. The only difference is that her internal source code is neatly organized, while mine is a tangled mess. There is one other difference: I sell mine for $6 and she sells hers for $10.

Since a customer never sees this source code, and it doesn't affect the operation of the app, why would anyone pay an extra $4 for Rebecca's software? Put more generally this should mean that it isn't worth paying more money for higher internal quality.

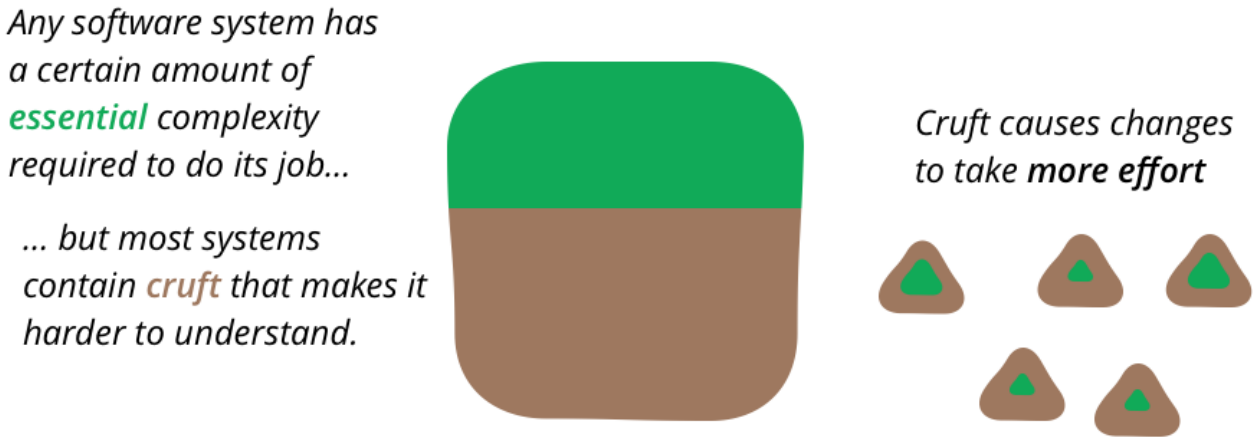
Another way I put this is that it makes sense to trade cost for external quality but it makes no sense to trade cost for internal quality. A user can judge whether they want to pay more to get a better user interface, since they can assess whether the user interface is sufficiently nicer to be worth the extra money. But a user can't see the internal modular structure of the software, let alone judge that it's better. Why pay more for something that has no effect? Since that's the case - why should any software developer put their time and effort into improving the internal quality of their work?

Internal quality makes it easier to enhance software

So why is it that software developers make an issue out of internal quality? Programmers spend most of their time modifying code. Even in a new system, almost all programming is done in the context of an existing code base. When I want to add a new feature to the software, my first task is to figure out how this feature fits into the flow of the existing application. I then need to change that flow to let my feature fit in. I often need to use data that's already in the application, so I need to understand what the data represents, how it relates to the data around it, and what data I may need to add for my new feature.

All of this is about me understanding the existing code. But it's very easy for software to be hard to understand. Logic can get tangled, the data can be hard to follow, the names used to refer to things may have made sense to Tony six months ago, but are as mysterious to me as his reasons for leaving the company. All of these are forms of what developers refer to as **cruft** - the difference between the current code and how it would ideally be.

A common metaphor for cruft is Technical Debt. The extra cost on adding features is like paying interest. Cleaning up the cruft is like paying down the principal. While it's a helpful metaphor, it does encourage many to believe cruft is much easier to measure and control than it is in practice.



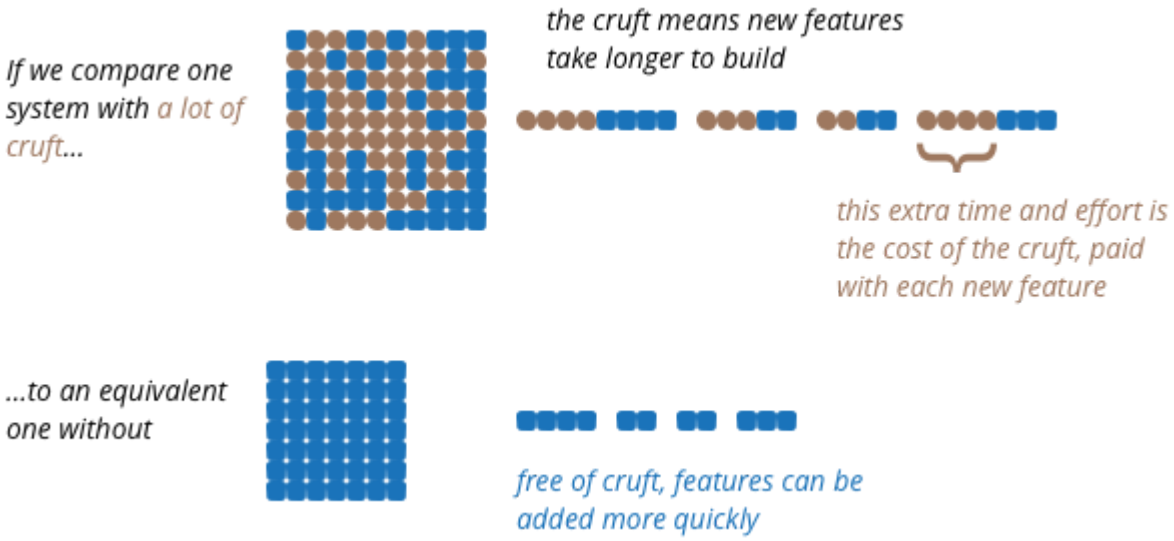
The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

One of the primary features of internal quality is making it easier for me to figure out how the application works so I can see how to add things. If the software is nicely divided into separate modules, I don't have to read all 500,000 lines of code, I can quickly find a few hundred lines in a couple of modules. If we've put the effort into clear naming, I can quickly understand what the various part of the code does without having to puzzle through the details. If the data sensibly follows the language and structure of the underlying business, I can easily understand how it correlates to the request I'm getting from the customer service reps. Cruft adds to the time it take for me to understand how to make a change, and also increases the chance that I'll make a mistake. If I spot my mistakes, then there's more time lost as I have to understand what the fault is and how to fix it. If I don't spot them, then we get production defects, and more time spend fixing things later.

My changes also affect the future. I may see a quick way to put in this feature, but it's a route that goes against the modular structure of the program, adding cruft. If I take that path, I'll make it quicker for me today, but slow down everyone else who has to deal with this code in future weeks and months. Once other members of the team make the same decision, an easy to modify application can quickly accumulate cruft to the point where every little change takes many weeks of effort.

Customers do care that new features come quickly

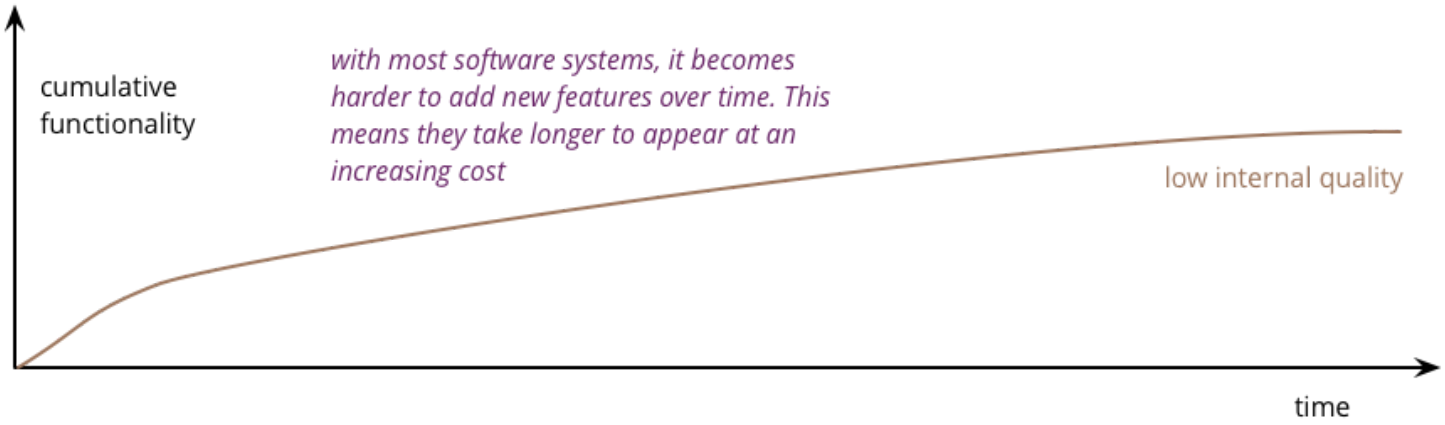
Here we see a clue of why internal quality does matter to users and customers. Better internal quality makes adding new features easier, therefore quicker and cheaper. Rebecca and I may have the same application now, but in the next few months Rebecca's high internal quality allows her to add new features every week, while I'm stuck trying chop through the cruft to get just a single new feature out. I can't compete with Rebecca's speed, and soon her software is far more featureful than mine. Then all my customers delete my app, and get Rebecca's instead, even as she's able to increase her price.



Visualizing the impact of internal quality

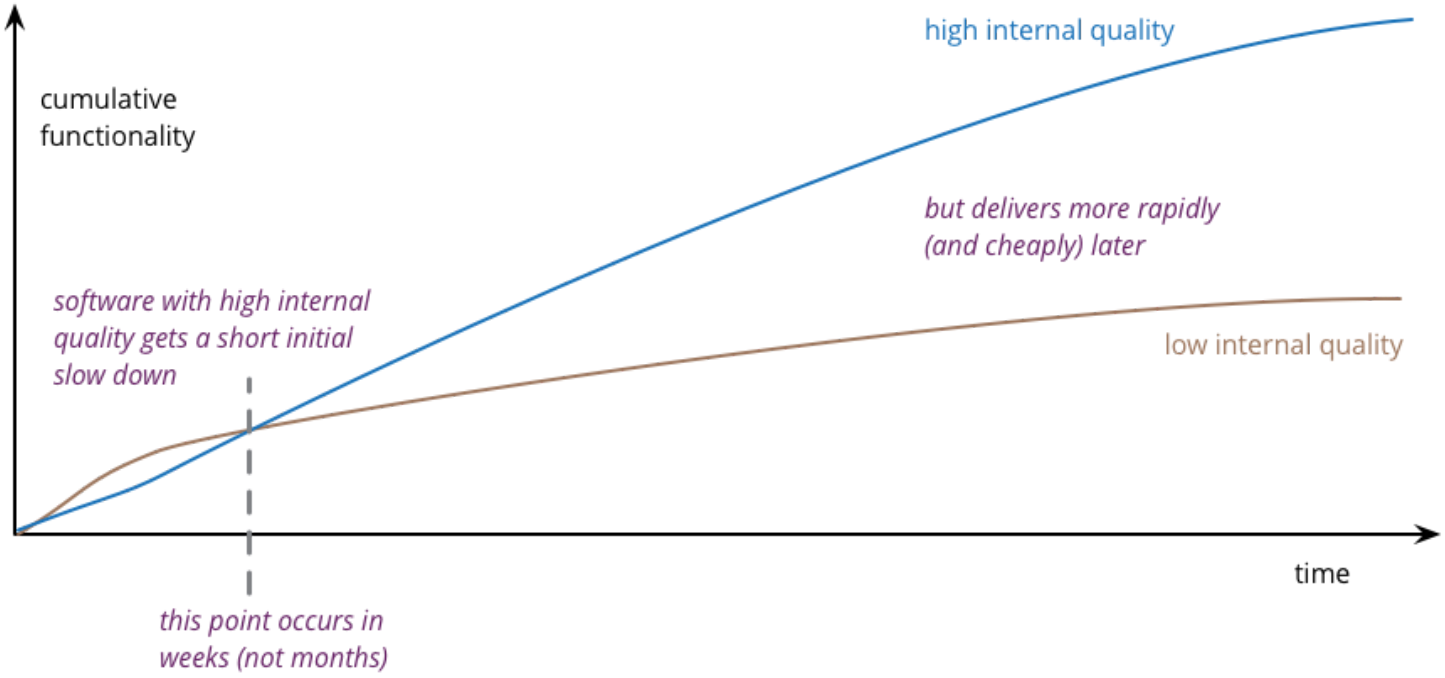
The fundamental role of internal quality is that it lowers the cost of future change. But there is some extra effort required to write good software, which does impose some cost in the short term.

A way of visualizing this is with the following pseudo-graph, where I plot the cumulative functionality of software versus the time (and thus cost) to produce it. For most software efforts, the curve looks something like this.



This is what happens with poor internal quality. Progress is rapid initially, but as time goes on it gets harder to add new features. Even small changes require programmers to understand large areas of code, code that's difficult to understand. When they make changes, unexpected breakages occur, leading to long test times and defects that need to be fixed.

Concentrating on high internal quality is about reducing that drop off in productivity. Indeed some products see an opposite effect, where developers can accelerate as new features can be easily built by making use of prior work. This happy situation is a rarer case, as it requires a skilled and disciplined team to make it happen. But we do occasionally see it.



The subtlety here is that there is a period where the low internal quality is more productive than the high track. During this time there is some kind of trade-off between quality and cost. The question, of course, is: how long is that period before the lines cross?

At this point we run into why this is a pseudo-graph. There is no way of measuring the functionality delivered by a software team. This inability to measure output, and thus productivity, makes it impossible to put solid numbers on the consequences of low internal quality (which is also difficult to measure). An inability to measure output is pretty common among professional work - how do we measure the productivity of lawyers or doctors?

The way I assess where lines cross is by canvassing the opinion of skilled developers that I know. And the answer surprises a lot of folks. Developers find poor quality code significantly slows them down within a few weeks. So there's not much runway where the trade-off between internal quality and cost applies. Even small software efforts benefit from attention to good software practices, certainly something I can attest from my experience.

Even the best teams create cruft

Many non-developers tend to think of cruft as something that only occurs when development teams are careless and make errors, but even the finest teams will inevitably create some cruft as they work.

I like to illustrate this point with a tale of when I was chatting with one of our best technical team leads. He'd just finished a project that was widely considered to be a great success. The client was happy with the delivered system, both in terms of its capabilities and its construction time and cost. Our people were positive about the experience of working on the project. The tech lead was broadly happy but confessed that the architecture of the system wasn't that good. I reacted with "how could that be - you're one of our best architects?" His reply is one familiar to any experienced software architect: "we made good decisions, but only now do we understand how we should have built it".

Many people, including more than a few in the software industry, liken building software to constructing cathedrals or skyscrapers - after all why do we use "architect" for senior programmers? But building software exists in a world of uncertainty unknown to the physical world. Software's customers have only a rough idea of what features they need in a product and learn more as the software is built - particularly once early versions are released to their users. The building blocks of software development - languages, libraries, and platforms - change significantly every few years. The equivalent in the physical world would be that customers usually add new floors and change the floor-plan once half the building is built and occupied, while the fundamental properties of concrete change every other year.

Dora studies on elite teams

The choice between quality and speed isn't the only choice in software development that makes intuitive sense, but is wrong. There is also a strong thread of thought that says there is a Bimodal choice between fast development, with frequent updates to a system, and reliable systems that

don't break in production. That this is a false choice is proven by the careful scientific work in the State Of Dev Ops Report.

For several years they have used statistical analysis of surveys to tease out the practices of high performing software teams. Their work has shown that elite software teams update production code many times a day, pushing code changes from development to production in less than an hour. As they do this, their change failure rate is significantly lower than slower organizations so they recover from errors much more quickly. Furthermore, such elite software delivery organizations are correlated with higher organizational performance.

Given this level of change, software projects are always creating something novel. We hardly ever find ourselves working on a well-understood problem that's been solved before. Naturally we learn most about the problem as we're building the solution, so it's common for me to hear that teams only really best understand what the architecture of their software should be after they've spent a year or so building it. Even the best teams will have cruft in their software.

The difference is that the best teams both create much less cruft but also remove enough of the cruft they do create that they can continue to add features quickly. They spend time creating automated tests so that they can surface problems quickly and spend less time removing bugs. They refactor frequently so that they can remove cruft before it builds up enough to get in the way. Continuous integration minimizes cruft building up due to team members working at cross-purposes. A common metaphor is that it's like cleaning up work surfaces and equipment in the kitchen. You can't not make things dirty when you cook, but if you don't clean things quickly, muck dries up, is harder to remove, and all the dirty stuff gets in the way of cooking the next dish.

High quality software is cheaper to produce

Summing all of this up:

- Neglecting internal quality leads to rapid build up of cruft
- This cruft slows down feature development

- Even a great team produces cruft, but by keeping internal quality high, is able to keep it under control
- High internal quality keeps cruft to a minimum, allowing a team to add features with less effort, time, and cost.

Sadly, software developers usually don't do a good job of explaining this situation. Countless times I've talked to development teams who say "they (management) won't let us write good quality code because it takes too long". Developers often justify attention to quality by justifying through the need for proper professionalism. But this moralistic argument implies that this quality comes at a cost - dooming their argument. The annoying thing is that the resulting crufty code both makes developers' lives harder, and costs the customer money. When thinking about internal quality, I stress that we should only approach it as an economic argument. High internal quality reduces the cost of future features, meaning that putting the time into writing good code actually reduces cost.

This is why the question that heads this article misses the point. The "cost" of high internal quality software is negative. The usual trade-off between cost and quality, one that we are used to for most decisions in our life, does not make sense with the internal quality of software. (It does for external quality, such as a carefully crafted user-experience.) Because the relationship between cost and internal quality is an unusual and counter-intuitive relationship, it's usually hard to absorb. But understanding it is critical to developing software at maximum efficiency.

▶ Significant Revisions