QUALITY / MATTERS

Quality software scales, lasts, and saves money.
This is how to win at it.

Acklen Avenue
Fall 2016

A/A

Existential Question: Does Quality Matter?

Should you care about the quality of your software under the hood? It seems like an odd question since, of course, we all care about quality. However, there may be times when someone might rightfully send quality to the back seat in the name of getting to the market quickly. In this case, all we need to realize is, we are pre-planning to re-write the software once funds stabilize.

But it might be a necessary evil if getting to market late means no company at all. Just so you know, that's not AT ALL what this article is about. Let's focus on your software project as if things have already stabilized and you're ready to slow down and do things right. Now, should you care about the quality of your software? We'll answer your question with another question: Do you like money?

Custom software carries a higher price tag. Ideally, the value that working software can bring should definitely overshadow the costs. If quality isn't on your radar, you might be surprised at the interesting dips your ROI graph can make. Fact: code quality is directly linked to the cost of custom software. Code quality affects maintenance effort which, in turn, affects your bottom line.

Maintenance costs are the problem. What's the solution? Well, we've already told you the secret: higher quality code. If you know how to get that, then you should contact me because we're always hiring! You can skip the rest of this article. Otherwise, keep reading.

80%
Of typical software costs is maintenance

40-60%
Of maintenance is a programmer interpreting existing code and identifying what is what

32-48%
Potential savings on overall maintenance costs with quality code

Why Isn't All Code High-Quality?

Coding is easy! We've all heard about 11-year-olds launching apps to the App Store and recent college grad's becoming "senior devs" and going to Silicon Valley to make millions. So, if coding is so easy, why do we hear such negative statistics about low code quality and high maintenance costs?

$150B

Poor quality software is estimated to cost the US economy over $150 billion annually

Well, we suppose the answer is, "Coding is easy, but building quality software is hard." Like most things in life, quality doesn't just "happen". It's something that takes intentional effort and dogged dedication from the first day to the last.

Developing software is a long series of small, seemingly insignificant experiments and decisions that lead to a working piece of functionality that hopefully works well with other parts of the system. To maintain high quality throughout the project, developers must constantly keep proven principles and practices at the front of their minds as they make those decisions.

If your dev team can make changes to your software, fix bugs, improve things, and generally make your software better over time without breaking the bank, then your code probably has a high level of quality. You have found some great software developers that really know what they're doing. Keep them on the payroll and give them a bonus.

How do I know if my software is high-quality?

Many times the only way to measure the true maintainability of a piece of software is postmortem. If your software is costing you way more than it should, and bug fixes or modifications take way too long, then chances are your software has a low level of quality and, therefore, is not maintainable.

On the other hand if your software developers move quickly, even in maintenance mode, and are able to fix bugs, modify things, and add new features quickly then you probably have high quality, maintainable software.

Developers know that the demand is high and they can work almost anywhere. If you're losing software developers and having to re-hire often, chances are your software is not maintainable.

Another tell of maintainable software is developer turnover. Developers who are working with maintainable software tend to enjoy their job more. If developers don't like what they're doing they will find another place to work. Developers know that the demand is high and they can work almost anywhere. If you're losing software developers and having to re-hire often, chances are your software is not maintainable.

Aside from the postmortem statistics for software maintainability, there are a few measures that can be taken during software development to help us understand whether software might be maintainable or not. One of those is “cyclomatic complexity”. This is a calculation based off of static analysis of your code that basically looks at several different aspects including lines of code, length of functions, links between objects, and other smells that typically impact complexity in a negative way.

When speaking of code, complexity if a bad thing. High complexity means lower maintainability, which also translates to lower quality (and higher costs, remember?). In fact, it has been proven that code with high cyclomatic complexity tends to have more defects.

Uh, can we know about that today, please? Well, this measurement can be taken from the first day to the last day of the life of a piece of software and can be displayed in a graph and easily trackable by everyone on the team. Cyclomatic complexity is “low hanging fruit” to help us understand whether software might or might not be maintainable in the future.

Another measurement of maintainability is test coverage. If test coverage is low, it is almost guaranteed that the code is not maintainable. If test coverage is very high, it (is not guaranteed, but) is likely that your code has a high level of maintainability for future developers. One reason for this is that the tests give quick alerts to developers when something has been broken inadvertently. That quick feedback cuts down on maintenance costs by shortening the lifespan of new bugs. Test coverage helps developers feel confident in making modifications, and it also helps provide some documentation to show developers how certain things should work, which will cut down on costly head-scratching. How can we get high-quality code?

Well, as we mentioned, writing maintainable code is hard. But thankfully there is a recipe that we have discovered in our own team that guides development teams to consistently produce high-quality, maintainable code. This recipe has been working for several years for our company and has helped to shape our company culture in really positive ways. We're proud to share our "secret sauce" with you!

1. BATTLE CRY

"Quality code or no code!"

Development teams must understand the impact of their work on the future. Like all of us, developers must put others before themselves. Developers need to understand what over-complex code, lack of test coverage, and overall unmaintainable code does to the client and to the life of the software. Once they understand what is at stake, they can begin to be on the same page, carry the same banner, and shout the same battle cry. The battle cry of a software developer who cares about maintainability is, "Quality code or no code."

A battle cry like this aligns a team on a common goal and helps keep each team member accountable to the others. A team that is united on this battle cry, even without other parts of this recipe, will be more likely to create more maintainable code than a team that is just going to let quality "happen". Creating maintainable code must start in the heart and must flow out from there. You need to desire to do something bigger than yourself.

2. PAIR PROGRAMMING

Pair programming serves to reduce the maintenance cost of software in so many ways including faster bug detection, better implementation and understanding of business requirements, and stricter adherence to clean code techniques, just to name a few.

Pair programming is a popular method for writing software but it usually drives project managers and financial types crazy. One keyboard, one monitor, one mouse, two chairs, two developers. It's one for the price of two.

Well, studies show that it's not actually one for the price of two. In fact the reason why pair programming is a part of this recipe, is because it yields almost the opposite. Remember, software development is not just from day one to first deployment. The cost of software includes development and maintenance all the way to the end of the life of the software.

Pair programming serves to reduce the maintenance cost of software in so many ways including faster bug detection, better implementation and understanding of business requirements, and stricter adherence to clean code techniques, just to name a few. Our company decided to start using pair programming as a rule since the beginning.

Pair programming has shaped our company's culture in many ways and has proven time and time again to lead us to much more maintainable software than solo development.

3. UNIT TESTS

If the presence of unit tests is a good measure for maintainability in software, then unit tests would naturally be a part of the recipe for creating maintainable code.

$100

Approximate cost of fixing a bug via unit test

Unit tests provide active, living documentation of the code, and provide a safety net for any modifications. They enable a team of developers to understand the code and how to modify it. They provide constant feedback on whether or not we are breaking things and other parts of the system.

$16,000

Approximate cost of fixing a bug after release

And on top of that, unit tests provide instant gratification to developers who love to see the fruits of their work as the day goes on (and not just after a few weeks). Any developer in his or her right mind would agree that unit tests are the right thing to do as a part of an overall software quality strategy.

So the fact that this is part of our recipe is really “candy on the bottom shelf.” Unit testing is not easy but it something that everyone knows they should do.

4. TEST FIRST

Ah ha! Here's the “gotcha.” Everyone can agree that unit testing is important to support maintainability, but not everyone agrees on when the tests should be written. Some would say that the test can be written after the production code to verify that the code works and to lock that code into a working state. Others say that the test must come first so that it helps to drive the design of the production code and also ensures that the code is testable and is more likely to have full test coverage.

Our recipe includes tests first. We have found that, by writing tests first followed by the production code, our code coverage is higher and our cyclomatic complexity is lower. Remember that cyclomatic complexity is one of the measurements for maintainable code.

Test first development is directly linked to lower complexity. Whereas test last development doesn't have an impact on cyclomatic complexity at all. The natural conclusion is to do tests first.

5. SOLID PRINCIPLES

The SOLID principles were assembled by an engineer named Robert Martin. He put these five principles together because they are a recipe, in and of of of themselves, for more “solid” code. Adhering these principles tends to help lower cyclomatic complexity and increase overall maintainability.

S	O	L	I	D
Single Responsibility Principle	**Open/Closed Principle**	**Liskov Substitution Principle**	**Interface Segregation Principle**	**Dependency Inversion Principle**

These principles work together to provide a holistic approach to writing maintainable software. The principles are very engineering-focused, but I’ll try to put them into terms that might make sense for the general audience. If you’re an engineer, you might want to skip this section since you might be offended by the over-simplification:

Single Responsibility Principle or "Keep everything small"

Now, that's not too hard to understand or see how that contributes to maintainability! The smaller something is or the less that it can do, the easier it is to maintain.

Open/Closed Principle or "Write things as if they will be replaced in their entirety later on."

This principle says that, when you have released something to production, you don't get to modify that thing ever again. Now, this is almost never followed to the letter, but by following this principle, we end up with more extension points and more modularity.

Liskov Substitution Principle or "Only allow connections when they should be made."

This means that we should never send a duck in a place where we should put a chicken. Your cell phone should never allow the wrong charger to be plugged in. Obviously something like this produces less side effects and, therefore, less bugs.

Interface Segregation Principle or "Keep things separate unless they are (really) similar."

Modularity again? Yes! Keeping things small that helps with maintainability. This principle leads us to divide functionality into logical sections and organize them into small units. Further, the principle helps us to remember that email senders should not also charge credit cards, for example.

Dependency Inversion Principle or "Outside forces should be interchangeable."

It sounds complicated, but really all this is talking about is interchangeability. If one month your software sends emails, and you decide that it should send text messages, you want those two things to be interchangeable without wrecking the rest of your software. Email or SMS senders are outside forces. They should be modular and ready to take a hike at any moment.

6. CONTINUOUS INTEGRATION

For software developers, daily development consists of anywhere from 10 to 100 uploads of code into a shared team workspace. Imagine if a large team was constantly pushing changes into a shared workspace without some sort of mechanism that checks for validity or conflicts or problems of any sort.

Remember that the longer a problem can exist, the more damage it does and the more time it takes to fix. The faster we can detect problems, the faster we can fix them. So part of our recipe to create maintainable software is having a system that checks our changes on every time we upload our code to our shared workspace.

Our system, for example, runs anywhere from 10 to 20 checks on our code automatically. In less than one minute, typically, are software developers get feedback on whatever it is that they just uploaded. If everything is “green”, they can go to the next piece of work. If it's red, they get instant feedback and know what to fix. This same system takes it a step farther and automates many of the mundane tasks that developers would otherwise have to do manually like deployment to a QA server. Continuous Integration keeps us honest and moving smoothly to the finish line.

OUR BEST & FINAL ADVICE

We just gave you 6 ingredients for a recipe that produces high-quality, maintainable software. Each ingredient is important, but none of them matter until you have a great team of people who care about the product they are creating. People who have the capacity to care about others as they code, design, or test are HARD to find. Developers who don’t mind sharing credit with their pairing partner are a rarity. To get truly high-quality software, it takes amazing people!

If amazing people like this are hard to find, we have to apologize... We might have hired them all already.