Issue 002: Quality

Graham Lee

November 5th, 2018

Welcome to the second issue of *De Programmatica Ipsum*, dedicated to the subject of *Quality*. In this edition:

- Adrian outlines his foolproof process for building a high-quality team[1].
- Graham tried to understand just what software quality is[2].
- Guest contributor Ioana Porcarasu explores the pursuit of quality[3] from waterfall to devops and beyond.

Enjoy this issue! Please let us know if you have any feedback[4] and get our free newsletter[5] to stay updated about new releases. If you want to support us, subscribe[6] for a month or a year, and let us know if you would like to write with us.[7]

Cover photo by VanveenJF[8] on Unsplash[9].

[1] https://deprogrammaticaipsum.com/the-ipsum-quality-process-12-steps-to-better-teams
[2] https://deprogrammaticaipsum.com/the-various-meanings-of-quality/
[3] https://deprogrammaticaipsum.com/on-some-things-quality-related/
[4] https://deprogrammaticaipsum.com/feedback/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/subscribe/
[7] https://deprogrammaticaipsum.com/write-with-us/
[8] https://unsplash.com/photos/xOS2ybxwxTI?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[9] https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

On Some Things Quality Related

Ioana Porcarasu

November 5th, 2018

Through the eyes of the writer

You never heard of me before, as this is my first article that "saw the public light". I don't have years and years of experience on my shoulders, but I can give you my thoughts, my personal opinions, and it's totally understandable if you don't agree with something, or even anything.

I started my journey into software engineering as an Automation in Test Engineer, around 7 years ago, and at that stage I had no idea what a complex concept quality was. During Computer Science university courses, we kept learning basics of programming languages, OOP, logic, maths, but barely close to nothing about what it involved into a product lifecycle. How do you get from one idea into an entire product available to customers, with a whole system on its back to monitor and track the users behaviour, to learn from their journeys and add improvements to it? And where is quality standing in these workflows? When are you confident enough in its capabilities to say: "Yes, we can ship this. Customer can have it!"?

I'm writing this article and keep re-reading, re-editing and re-doing it. Every time, I keep adding something, a small improvement hopefully, but no one else had read it except myself. All I do is updating it all over again, but with no external feedback. So how can I know that my "product" meets its minimum level of quality? How can I know it meets your needs and expectations?

So, what is quality?

The most basic definition of quality can be identified as grading how good or bad something is. But that doesn't necessarily mean that "your bad" is also "my bad" , or the other way

around! Quality is a subjective attribute, with different meaning to different people[1].

Quality, in essence, is embedded in our everyday routine: we are looking to buy a new gadget, so we search reviews for different products to asses their quality before actually purchasing one. We read positives and improvement points for different brands until we have the confidence that one of them can actually meet our needs, our expectations. We are researching for our next holiday, so we browse countless websites and brochures until we find the one that fulfils our requirements. So our entire live is actually rolling around this concept of quality.

And this can be seen in product development as well. We make assumptions all the time related to what we think is right for the customers. But quality should mostly be seen through the eyes of the customer.

Software quality is defined as the level to which a system meets specified requirements or user needs and expectations. Testing has become an important segment in the software development process to ensure its quality.

One of the basic testing principle is related to verification vs validation. Verification is testing that your product meets the requirements written for the product: "Did we build what we said we would? Did we build the system right?". Validation, on the other hand, tests how well you addressed the business needs that caused you to write those specifications: "Did we build what the customer actually needs? Did we build the right system?".

Ever since the oldie, waterfall methodology, we understood that we didn't actually provide quality services. We could work for a year on a new product; research done, implemented this awesome new idea; tested and validated it for a couple more weeks, and finally delivered it to our customers in a shiny new box with a red bow at the top; if the product is not fit for purpose for the customer needs, what can we do next? Can we say anything about the quality of that product? On one side, we verified it extensively during our (let's say) 6-week window and reported appropriately, it met the requirements and we were highly confident in it! We build the system right, but coming back to the previous point, was it actually the right system? Even if the idea might've been validated initially, in the year of developing and making the product available to the customer, he might've changed their priorities, he realised that actually another thing is needed, another problem needs solving . So, providing quality enforces this idea of building the right system for the user. Feeding back to the customer more often was essential to understand if we're providing quality services and eliminate waste.

And that's one of the reasons why software development evolved into all the new different trends happening currently.

One definition proposed by Jez Humble is that DevOps is "a cross-disciplinary community of practice dedicated to the study of building, evolving and operating rapidly-changing resilient systems at scale.". Continouous delivery concept is also seen as "the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way." The entire community understood that we can use different techniques, we can collaborate more and we can help each other to mitigate risks and get the confidence that we are delivering value to our customers, and all this through small incremental changes.

We are more focused now in working in cross-functional teams that broke down the silos and the fences, and we can give feedback as early as possible to improve the quality of the system. Just by asking questions when analysing a set of requirements challenges the group

[1] https://deprogrammaticaipsum.com/the-various-meanings-of-quality/

of thinking at different scenarios, improving collaboration between developers, testers, and operations. And how much does that cost? Nothing! We didn't even start implementing it!

And quality doesn't cover only the system anymore. It evolved as a whole new process, and we moved away from testing only roles, but to Quality Engineering ones. And as the Testing manifesto is stating, we are not focusing only in proving a level of confidence for the product anymore, but we actively collaborate towards:

- **Testing throughout OVER testing at the end** – As we've seen in so many examples and memories, the amount of waste increases exponentially if we leave the testing at the end of the product lifecycle. Instead, we should focus our efforts in testing early and testing often. And with the massive help of automated checks, it all comes back to fast feedback loop!
- **Testing understanding OVER checking functionality** – Similar as above, why waiting to get a new build to test and confirm behaviours, when some of the possible issues could be found in earlier stages? Shifting left the testing is a massive improvement to the overall quality of the system as it might raise questions in team's understandings or even customer expectations. Building a common understating will automatically lead to improved quality.
- **Preventing bugs OVER finding bugs** – This is essential if we want to respond fast to our customer needs. We need to think more about the "life" after releasing to the customer. How can we prove it is actually working as expected and if anything happens, we can act fast?
- **Building the system OVER breaking the system** – We need to be proactive and work closely with the engineers, POs to identify what the customer actually wants. We need to steps into theirs shoes and help understanding the problems we are trying to solve for them and find the best solutions.
- **Team responsibility for quality OVER tester responsibility** – Every single team member is responsible for quality of the product we are building.

Testers do not provide assurance anymore; they help with analysis and feedback. With every engineer writing and running tests with their commits, they show ownership. With the whole team caring about the state of the system or of the build, they show ownership. With good monitoring and logging in place, the team then owns quality. Through a continuous-improvement mindset applied throughout the lifecycle of a system, we demonstrate that ownership of quality. And the truth is that shared ownership results in higher quality!

—

Photo by Brian Goff[2] on Unsplash[3].

[2]https://unsplash.com/photos/G-dfxw3DFYc?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[3]https://unsplash.com/search/photos/waterfall?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Various Meanings Of Quality

Graham Lee

November 5th, 2018

What, to you, is quality software?

I'm sure that I will get different responses from different people. Quality is not absolute, and what you consider high-quality may be irrelevant to somebody else[1]. Your view of high quality may even be a sign of poor quality to another reviewer. Let's take a look at some aspects of software quality.

Code Quality

Developers, myself included, often think of the quality of the software's source code: is it easy to understand and change? Does the code use paradigms that I find admirable? Does it use functional programming, for example, if I'm in to that sort of thing?

Did the author write automated tests? Do the tests provide good coverage? Are they appropriately distributed between different levels of granularity? Do the "grains" (the units, modules, or subsystems) represent useful models or abstractions?

Is there documentation? If there is not, is it because the code is self-documenting? Is there a single command to type or button to press to get a production build out? Does committing to the repository automatically trigger a production build?

[1] https://deprogrammaticaipsum.com/on-some-things-quality-related/

Does Code Quality Matter?

These questions all relate to a particular perspective on the quality of software, but we must be honest with ourselves. For the most part, customers considering whether to license a particular software product or subscribe to software as a service very rarely even *look* at the source code, let alone make an informed judgement as to its quality. In most cases, they aren't even allowed to look.

Even in the case of free software or open source products, the likelihood of spy-before-you-try is low. I cannot bring to mind the thousands of such projects I must have used during my career to date: Eclipse, LibreOffice, Linux, the GNU Compiler Collection, Darwin, clang, LaTeX, Firefox, FreeBSD, GNU Emacs, node.js...and I can't think how vanishingly small must be the fraction of those of which I have even *partially* examined the source code. Probably not even one per cent.

Lemons

Now that means that if the quality of a software product, or service, *is* related to the code quality, then there's a lemon market[2] for software. People can't (or don't) evaluate the quality of the code, so won't pay more for premium-quality code (and hence software) than they would for a clunker.

(As an aside, the solution for the lemon problem in motor vehicles is to provide a warranty. In software, commercial and free software alike is provided "AS IS", all in capital letters, with no suggestion that the maker expects it to work the way you like, or even the way they said in the marketing materials.)

Zen and the Art of Software Maintenance

The quality of *the code* in a software system is an aspect of its internal quality. Borrowing an analogy from Robert M. Pirsig[3], interest in this form of quality is like being interested in the internals and maintainability of a motorcycle engine. It's a "classical" form of quality, in which one must be mindful of all the workings and details of the system under consideration.

Other people will have other ideas of software quality, perhaps informed by the "romantic" school of quality.

To Accessibility and Beyond

Is the product accessible? That may mean that the controls are suited to full keyboard navigation, text-to-speech readers, joystick control, or other adaptive input and output schemes. Accessibility also includes ensuring that the colour scheme used through the UI is suitable for people with colour vision deficiency[4].

Some would say that accessibility is one aspect of the wider principle of *usability* and consideration of the user experience. Can the capabilities of the software be discovered easily? Can they be used efficiently? Do the capabilities that a given person requires match their expectations? Do those capabilities even *exist*?

[2] https://www.investopedia.com/terms/l/lemons-problem.asp
[3] https://www.goodreads.com/book/show/629.Zen_and_the_Art_of_Motorcycle_Maintenance
[4] http://www.colourblindawareness.org/colour-blindness/

Other External Qualities

We could carry on listing other external "qualities" of a software product or service: security is a very big topic and indeed the next issue of de Programmatica Ipsum[5] will focus exclusively on that.

Performance, both in terms of speed of computation and responsiveness (a researcher running their scientific codes on a high-performance computing installation will experience very high latency, as the queueing system could take days to even start their job; they may not think of this as "unresponsive" if submitting to the queue happens quickly). The resource usage is another form of quality, as is the clarity of the customer documentation or even the helpfulness of the customer support team.

How We Plan For These Qualities

Many of these quality attributes will be described by most people on a software project as the "non-functional requirements" and batched together as things we should probably keep an eye on while planning to construct the functional requirements. Meanwhile, the software architect (if there is one) will be trying to design a system that satisfies all of the non-functional requirements while paying half a mind to the question of whether it can satisfy the functional requirements.

Which finally, nearly eight hundred words into an article on software quality, brings us to the elephant in the room: does the software even *do* the things people want of it? That seems like an important measurement of software quality, too.

The Quality of Correctness

Does our software do what our customers want, or need? Manny Lehman tells us[6] that in many cases the answer to this question is not stable. You are not making it up and your customers are not deliberately being awkward, their needs genuinely are evolving. Perhaps, in a very postmodern way, the trigger for their evolution was the introduction of the software itself. The availability of version 1 freed up some time, so that they realised they could do these other things if only they had version 2.

Given the evolutionary nature of software requirements, how should we consider the "quality assurance" function in a software team? For a start, let us rid ourselves of a misconception: your "QAs" do not assure quality. At best, they *measure* quality by testing (or creating automated test scripts for) your software. But testing does not assure us of quality. As Dijkstra said, it can tell us when bugs are present, but not when they are absent.

Regression Tests

Most of the software projects I've worked on have had a battery of "regression tests". Only some of these were written to detect regressions. Most were written while TDDing, or to demonstrate that a feature worked as part of that feature's construction: they are accretion tests, rather than regression tests.

A *true* regression test would be one where we know that the behaviour must not be reverted. But that doesn't mean that we should test *anything that was ever added*, it should mean knowing *what your customers are using today* and how bad it would be if it were to break.

[5] https://deprogrammaticaipsum.com
[6] https://ieeexplore.ieee.org/document/1456074?reload=true

"Enough" Quality

Notice that I said *how bad*, as there are different amounts of badness. The Joel Test[7] asked whether we always fix new bugs before we write new code. That doesn't seem like a good way of working. Success in business is always about opportunity, and so we need to consider *opportunity cost*. If three people would be put off our product by a minor bug, and thirty people would buy the product if we added a minor feature, we *should* write the new code before fixing the bug. On the other hand if the bug breaks every day features for every customer of our product, then yes, it's time to fix it.

There are lots of ways to evaluate the quality of a software product. Now what, to *your customers*, is quality software?

— Photo by Dane Deaner[8] on Unsplash[9].

[7] https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/

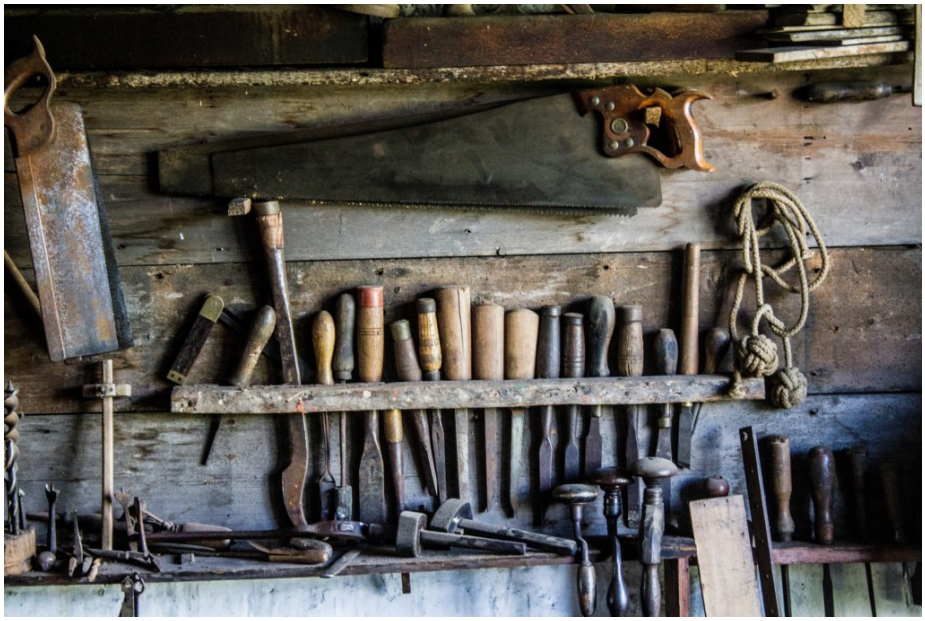
[8] https://unsplash.com/photos/7woHBtwCgTQ?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[9] https://unsplash.com/search/photos/lemon-market?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Ipsum Quality Process: 12 Steps To Better Teams

Adrian Kosmaczewski

November 5th, 2018



Back in 2005 I had an interview for a job in a software consulting company in the French-speaking part of Switzerland. The interview in itself was dreadful; the hiring manager was certainly more interested in the "gaps" in my résumé than anything else. But one thing drew my attention, and made the interview nevertheless memorable; this person boasted that their company was one of the few "certified CMMI level 2" in Switzerland, and that they were in the process of preparing for the "level 3" certification.

To be frank I had no idea what this person was talking about; so I looked online and discovered that:

> Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program. (...) It is required by many United States Department of Defense (DoD) and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization.

— Wikipedia[1]

[1] https://en.wikipedia.org/wiki/Capability_Maturity_Model_Integration

Methodologies

The important word in the quote above is "claims." CMMI is just another quality framework, just like ISO 9000[2], Six Sigma[3], Total Quality Management[4], or even the Joel Test[5]; the primary job of these frameworks is to provide a nice (and sometimes very expensive) badge for companies to feature in their website, getting C-level managers to tap each other on their shoulders and give themselves another huge bonus.

A quality framework does not, per se, provide any insurance against big balls of mud[6]. It does not establish any expected level of quality. It provides no insurance against impossible deadlines, crazy customers, developers with family issues, computer failure, misleading vendor marketing, or anything else. Yet, companies all over the world subscribe to these and other frameworks, creating an industry of speakers, consultants and conferences worth billions, yet unable to answer The Real Question That Everybody Asks About Their Software Project™®©:

> Will the project be finished in time, delivering the features the user needs, and within budget?

If these frameworks are of no use, what is then, the magical factor that separates the (few) companies releasing quality software in time (there are, believe me) compared to the rest? Why do most projects end up with a Death March[7] instead?

About Quality

I tend to describe Quality with a simple phrase, almost a mantra; a phrase very easy to remember, borrowed from a book somewhere, which I will shamelessly transcribe here for you to invoke *ad nauseam* in your next meetings; répétez avec moi:

> Quality is doing the right thing, and doing the thing right.

(As the astute reader has guessed by now, the quality frameworks discussed in the previous section only care about the second part of the statement, if anything.)

But what is the *thing* we have to do, and rightfully so? Well, in the case of software engineering, that "thing" is *not* software – and I am pretty sure I have just made a few eyebrows raise in perplexity. No, ladies and gentlemen: the "thing" in question is the following:

> To take care of your team.

There you go. Here is my invoice. Welcome to the light. There is no other thing to know.

Quality is taking care of your team. A product with good quality is simply a side effect of a team being taken care of (sorry for the functional programmers among my readers.) Do you want quality in your software product? Take care of your team. (Hey, Graham[8], we should start a software consultancy and copyright this idea fast.)

[2] https://en.wikipedia.org/wiki/ISO_9000
[3] https://en.wikipedia.org/wiki/Six_Sigma
[4] https://en.wikipedia.org/wiki/Total_quality_management
[5] https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/
[6] http://laputan.org/mud/
[7] https://www.amazon.com/Death-March-2nd-Edward-Yourdon/dp/013143635X
[8] https://deprogrammaticaipsum.com/user/graham/

The Ipsum Quality Process

And what is, exactly, taking care of a team? I am glad you asked. Taking care of a team can be defined as a set of 12 simple bullet points that you can start following right now in order to make better software:

1. Hire inclusively

Having members with disabilities is important for your team – if anything because, among other reasons, it can help make your product more accessible.

Having team members from other cultures is important for your team – if anything because, among other reasons, it can help make your software multicultural.

Having people with various ages is important for your team – if anything because, among other reasons, it opens up opportunities for coaching and mentoring.

Having people from all genders is important for your team – if anything because, among other reasons, it will make the world a wonderfully better place.

All of this is important for your team, and hence it is important for your product, for the industry, and for the world.

2. Pay your engineers decent and equal wages, including extra hours

The software engineering field is extremely profitable, and it is not unheard of for a consulting company to have margins well above 30 or 40 percent. Why is it then that only managers get bonuses at the end of the year? Why do not these companies pay extra time? Why is it that the salaries of software developers are going down?

One of the main reasons of this situation is the lack of unions in our field; the bargaining power of software engineers is scattered, atomized, which was exactly the objective of the managers of businesses in the first place. As a developer, you must ask for a clear, open salary policy, and make sure all engineers are being paid in equal terms in correlation to their experience, regardless of age, gender, nationality, skin color, or any other attribute.

Regarding extra time, if it must happen for any reason, *pay it* and then *compensate it* with extra days for rest (see point 9 below.)

3. Stop cramming people in open spaces

Get private offices for your team, with at most two or three people in them. Forbid the use of smartphones and Skype and Hangouts in those rooms; they are there to be used for those three or four hours of deep concentration that make projects move forward fast (see point 9 below.) Set up many team rooms with whiteboards so people can discuss and talk. Isolate all of those rooms acustically from one another.

You need to coordinate, yes; but not as often as you need to get things done. Do not say that private offices are more expensive, because they are not – at least not in the long run. Have a broader mindset and stop telling lies.

4. Do not offshore projects

Offshoring software projects, that is, outsourcing them overseas, is the 21st century equivalent of economic imperialism, keeping countries in Latin America, Asia or Africa earn-

ing peanuts while agencies fill their pockets, and blocking the development of those countries. Emerging economies should be building their own products, and developed economies should pay for those products. This is the way to development; not imperialism.

All countries need software engineers to stay where they are, if anything because their salaries feed a greater economy. And also, the trouble and the mess of managing a project overseas is actually going to eat all the benefits of having a team located next to you, so do not even think about offshoring.

5. Train teams in old and new technologies

Give everybody in the team a certain budget to spend in conferences, online courses, and reading material. Make a library of physical books and stack it with some timeless classics. Give your team clear learning objectives, such as getting certified, preparing a 20-page document to share with the team, or talking at some event, and set up bonus structures tied to those requirements.

Make them teach each other all of those new things, continuously; either in pair programming sessions, either during code reviews, or in small 20 minute talks in the conference room in front of the whole team (which is a good exercise for people new to public speaking.) Review that your team has learnt and look up to the horizon. Rinse and repeat.

6. Drop the foosball, ping pong, pool table, karaoke, game console, ...

Seriously. It only shows that all you know about developers is what you learnt watching episodes of "The IT Crowd."[9] Those things are not only annoying, they are vexing and demeaning.

Build a library instead. With books, you know, the dead tree kind, and some coaches; a water boiler and some tea; quiet music, because relaxation is key. Offer a place for relaxation and your team will love you. Peace and thoughtfulness will make your team create better software. Certainly not foosball tables.

7. Watch for mental health issues in your team

Mobbing, burnout, harassment, depression, fatigue; all of these things[10] are happening right now in your team. Oh yes, they are. Are you even paying attention?

8. Watch for body health issues in your team

Does your team have standing desks? Do they have good lightning? Do they have air conditioning in summer and heating in winter? Do they have to use noise-cancelling headphones every day? Do they have good monitors to prevent strain in their eyes? Are they sitting in confortable chairs? Do they take enough pauses? Do they use ergonomic keyboards? Are they doing extra hours? Are they eating in front of their computers? What are they exactly eating?

9. Plan for 6 hours of actual work per day

Each workday should be divided as such:

[9] https://en.wikipedia.org/wiki/The_IT_Crowd

[10] https://www.creativebloq.com/advice/10-steps-to-protecting-your-mental-health-at-work

- 4 hours a day of actual work (coding, writing documentation.)
- 2 hours a day of mingling, drinking coffee, lunch, meetings and coordination.
- And 2 hours a day for learning something new.

More than 6 hours of actual software development work per day is a recipe for disaster, burnout, and turnover (been there, done that.) If your work laws mandate 40 hours a week of work (which is the case in Switzerland,) make sure that those extra 2 hours per day are spent learning something new instead of writing more code (see point 5 above.) Send your team members home at the end of the working day. Turn off the electricity in the office if needed, so that they actually leave. Compensate for extra time (see point 2 above.)

10. Embrace remote work

Remote Work Works™®©. And beautifully so. Our field is perfectly suited for working from home. Embrace it and drop the requirement for command and control. You are going to make economies, and your employees are going to love you more for this.

11. Do not be a project management methodology fundamentalist

You do not need to have a scrum standup meeting every morning. You do not need to have that retrospective every two weeks. You do not need to follow the PMI[11] workbook to the letter and punish those who challenge it. Doing so is nothing else but cargo cult.[12] Remember this:

> Waterfall has put a man on the moon. What has your methodology done so far for your team?

12. Change your approach to job interviews

Stop asking about round pot hole covers. Stop those live coding challenges to reverse a linked list. Drop the academic requirements. Instead, hire for character first, and then for skills.

Of course you need to be sure if the candidate can do the job, so here is a simple technique: *hire them to work in your team for a day.* Just a day. Pay them a full, fair freelance rate (see point 2 above) and watch them evolve in your team. At the end of the day, ask yourself and your team these questions:

- Were they good at reading code?
- Were they nice people?
- Did they suggest improvements to the code?
- Did they ask good questions?
- Did they behave ethically?
- Did they say "I do not know" often enough?
- Were they humble?
- Did they show empathy and kindness towards other team members?
- Did they looked happy being there?
- Did they ask about code coverage or the CI setup?
- Did they offer to help, even if the time was short?
- Did your team come back to you with big smiles asking you when they were going to see them again?

[11] https://www.pmi.org/
[12] https://en.wikipedia.org/wiki/Cargo_cult

If most answers were "yes," just hire the person.

Disclaimer

If you are a project manager or software CEO[13] reading this article, you must be asking yourself the following question: does the Ipsum Quality Process guarantee the outcome of my project?

No, of course not. You can still have incompetent people making bad decisions, even if they are healthy, well paid, and enjoying excellent work conditions. But at least you will have reduced your employee turnover; you will spend less money in hiring costs; employees will have less sick days; they will be proud of working in those teams, and will attract their friends; they will end up doing everything they can for the company, including, why yes, writing high quality code yielding high quality products.

And I am not the only one who thinks like this.

> Take care of your employees and they'll take care of your business.

— Richard Branson[14], Founder of Virgin Group.

In short: only after your company fully implements the 12 steps above, we can sit down and talk about TDD, Agile, CI and whatnot.

Conclusion

To finish the anecdote that started this article, I heard the company (which shall remain nameless) went almost bankrupt, after taking in a project much bigger than they could chew, having it developed by an offshore team working in an open space. Most of their technical crew left the company less than half a year later. I read that they struggled to survive, and they ended up sacking most of their management and enduring a deep restructuration.

As for the CMMI level 3 certification, a quick look at their current website tells me they never got it, or they just do not care about it anymore.

> "Move fast, break things and piss off the ones taking over" ™©®

— Eliezer Talon[15]

Cover photo by Philip Swinburn[16] on Unsplash[17].

[13] https://twitter.com/PHP_CEO
[14] https://www.virgin.com/entrepreneur/why-is-looking-after-your-employees-so-important
[15] http://twitter.com/elitalon
[16] https://unsplash.com/photos/vS7LVkPyXJU?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[17] https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText