Software quality without testing Part 1

When I started to write this article, I thought of doing a transcript of the talk we held at We are Developers World Congress 2019. Half a day later, and five pages into this article, I thought to myself: “No one will ever such a giant wall of text!” So, I decided to split that article into a series of three.

In Part one, we cover Quality in general, and what we actually mean when we say “software quality without testing”: We talk about what we feel is the reality about quality, and what quality means for us. Therefore, we go deeper into quality characteristics and the definition of “quality” itself.

In Part two we'll cover the responsibilities of each team member: Where should quality for each member start, and where should it end (or should it end at all?) We introduce you to the quality hopper, which is a visual aid of how quality could be structured, and we will also go over the topic of requirements, user-stories and acceptance criteria.

In Part three we'll cover categories of projects and how to select the right project type for your needs. We’ll talk about how to setup a quality friendly technical environment in which quality can be already established and delivered before we even move towards testing the software.

Software quality without testing? Possible!?

Before we dive into this, let me just say the title of this article series is a blatant click-bait. As usual, Betteridge's law of headlines applies: *Any headline that ends in a question mark can be answered by the word “No”*. Right from the start I want to apologize for any confusion. There is no such thing as software quality without testing. It would be the similar to talking about implementing software without translating the requirements into an executable form (code or otherwise), which is simply not possible (we are not talking about MDA’s or software factories, even when the still use code which has to be assembled by the machine).

Thinking about quality means for most, thinking about testing. Either low-level tests like unit- and integration-tests or high-level tests like (user)acceptance-test UI- or regression-tests. But quality should be (and is) more than just testing your software. Quality should be an organisation wide unified mindset; each role and each member should have a “quality-first” attitude, even if their perspectives and opinion on this topic differ.

The thought about a quality mind-set seams natural since we are striving to build a better product, but the reality, how people use the word quality and act during the software lifecycle is something else entirely.

The truth about Quality

As we work for different clients in different projects and various industries, we often hear the same "problems" about creating quality. Just to name a few we collected some, generalized them and give you our thoughts to these quotes.

> We ensure quality because we provide automated tests.

You test the software by automatically checking if it exhibits an expected behaviour. Sure, this is important in our fast-paced world. We need all sorts of automated tests; software grows more complex by the day. And within its complexity the need for Quality-First thinking is absolutely necessary. Testing the software with anything else then TDD or BDD means you test the software *only* after its implementation. And this is where

it’s already expensive to find and fix bugs, especially in architecture and structure or its requirements. So, you provide automated tests - but that doesn't mean you’re thinking about quality.

> It’s too expensive and takes too much time to write unit-tests.

A developer’s work is done only after his/her software is tested properly. We understand that TDD is not always possible. However, thinking about writing unit-tests later mostly equals to never actually doing. Writing unit-test that test your code properly with all its complexity a couple of days or even weeks later seems to not align with the priorities in most teams, and it also appears just not be human nature. Implement unit-test while the train of thought is still fresh so you could cover most of the complexity of your code. Estimate your stories and tasks - including the implementation of unit-tests. Don't ask, just do. We often hear the stakeholder don't want to pay for unit-tests. The truth is, the stakeholder often doesn't know what that means. In our experience a stakeholder will not be mad at you if you deliver reliable and maintainable software.

Did you ever hear the statement this part of code is not testable? Start thinking about redesigning this part. Software, which is not testable can be more hassle than it is worth (not to mention the usually astronomical cost of maintaining it or, worse, working around its limitations).
The reason you should write meaningful unit-tests is obvious if you think about it with quality in mind. Unit-tests will help you maintain, extend and improve your software. Refactoring will not be such a huge pain. With the help of a good unit-test-set you will easily see where your changes break something, and where software behaves in a non-expected way. Your unit-test set should cover at least 60%-80% of your code. However, these numbers are only relevant if your unit-test tests are testing the software in a meaningful way that actually covers and reduces risks.

> We’ve got qualified testers, who verify that our software behaves as expected.

Most projects and clients do have good testers or (at least testers), that is true. We also see all the time that this QA-Engineers are used for regression tests or other monotone tasks. So, the QA-Engineer is used as a click-monkey - sentenced to do the same task repeatedly. It boring, it's error prone because when doing repetitive things start getting blind to unexpected behaviour, and it's a waste of qualified resources. Again, invest in test-automation for these tasks: a QA-Engineer should focus on exploring and breaking new features testing, before the feature goes live. This is where their qualification is needed; but it’s also where their skills & passion truly shine. A QA-Engineer should find bugs. Test-automation should verify that its behaviour did not change in unintended ways. And no QA-System - neither test-automation nor QA-Engineer - will be able to test quality into the system under test. The QA-System will only show where there is a lack of quality.

> It’s just a Proof of Concept, but we would like to use it productively as soon as possible

This is one of my favourite quotes/situations. In Part 3 we will go into the idea of project types and how to define a project. Let’s think about the requirements and the expected outcome: In this case, maybe a PoC is not enough for your what you intend to do with it.

So, what is “quality”? Sales and delivery often use it as a buzzword, like **we deliver high quality software** - but what exactly does that mean?

The definition of quality and software quality by ISTQB is the following:

> **quality:** The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

> **software quality:** The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

So even the ISTQB refers to function and technical point of view of software quality. We, however, want to talk about the mindset. And therefore, it's important to define our understanding of “quality”.

Quality Characteristics

The Quality building blocks

Like with any other conceptual thought, we need something to base our theories on, so we can build on a solid foundation. Take a tree for example. You can throw around some seeds and hope you'll get a tree someday, which, in a city like Vienna, is quite unlikely. It’s more likely that the next pigeon will be snacking away happily on the seeds you so thoughtfully provided for it.

Plant the seeds in good soil, water the them, fertilise it, and you will get a tree, just like you expected to. It is not instant. It takes care and work. The same goes for quality. Throwing around some test automation, or some part of a CI/CD pipeline will gain you nothing except frustration. Things might work individually, but they won’t play well together, and overall, you’ll engage some poor engineer who has to spend hours maintaining and "improving" a shaky construct that is not built on a solid foundation. Most likely, the result is worthless and expensive. Like the tree, quality needs structure and order to grow.

The Team

Our fist building block is kind of obvious. It's the team, and how we work and interact with each other. A forming and storming team will not produce overall good quality. To build a team you need trust, team players, a common vision, and attainable goals. Don't set your goals too low, this will bore the team. But don't set goals unachievably high either. This will demotivate the team and the performance will drop, from not having any sense of achievement.

The forming and norming phase of team building is where the team should learn how to set goals for themselves in a meaningful way. This will help to transition into the performing phase, where a team might produce magic results over long distances. This leads us almost seamlessly to the next building block.

Communication

A team – and in fact the whole organisation - is nothing without good communication. Communication has to be learned; teams and their leads, as well as management, should do internal and external workshops somewhat regularly. Methods like BDD can help to simplify the communication between stakeholders, PO's and the development team. This leads to better understanding of the different points of view, and also minimizes social friction because of a common understanding and language.

Metrics

Metrics and monitoring are important - no argument there. We need this to measure what we call objective quality.

- It will give us the possibility to define quality gates in a CI/CD pipeline
- Static code analyser and its KPI's help us to identify architectural- and code-smells – including, but not limited to:
    - Code-Duplications
    - Circular references
    - Coding Guideline violations
- We will gain a better understanding on how the software performs
    - Performance-tests
    - Load-tests
    - Mean time to recovery/repair (MTTR)
- But also gives us the KPI's we need to say our software is well tested
    - Code-Coverage (Unit-tests)
    - Path- Coverage (Unit-, Integration test)
    - Number of defects/bugs found

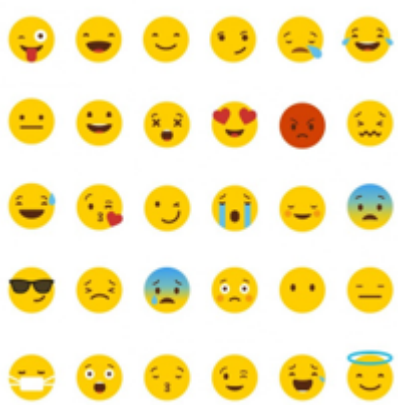
These metrics look mightily pretty when they are displayed on a monitoring system, but they can also give us...

- ... a false sense of security.
- ... a wrong picture of the actual state of the software if ...

- ... the tests considered for code-coverage are not meaningful tests.
- ... the tests are only testing the happy path.

So objective quality is important, yes. But its meaning and its value should be critically scrutinised to make the most of it.

Emotions



Human beings are emotionally driven. We tend to “feel” that something is good or bad. Obviously, this has a lot to do with demographics, ages, origin, childhood and social situation. Just think about "iOS vs. Android" which is almost a religiously driven discussion. Take a step back, you'll see that both OSes do basically the same, they even run the same Apps, and both OS’es are of high quality.

The same goes for roles in a project. The stakeholder has a different perspective of what is important, and therefore what “good quality” means than an end-user, or a developer. This all needs to be considered when we ask them about what they think (better: feel) is “high quality”. And this also heavily affects our perceived value of the system we’re looking at. Since no sane human being is above these emotions, subjective quality is something we must deal with as well as possible, on a daily basis. (btw. Android Rocks 😃)

Summary

Quality is a mindset. Sure, there is a technical implementation of a certain quality, with all the tests a QA-Engineer may provide, and all the metrics we can capture and generate informative KPI's from. but overall, we need to shape our thought processes towards quality right from the beginning of every single project. What flight attendants always tell us also applies for a quality-oriented mindset: “Put on your own masks before helping others”: As a part of a team which implements and deliver software it's your task and responsibility to form your quality-first mindset, and also shape the mindset of your team members second.

In Part two we'll cover the responsibilities of each team member. Where should quality for each member start? And where should it end (or should it end at all?). We introduce you to the quality hopper, which is a visual aid of how quality could be structured, and we will also go over the topic of requirements, user-stories and acceptance criteria.