

Software quality without testing Part 3

In the first part of this three part series we learned about the importance of a quality mindset or better said, a quality-first mindset, also we looked into our building block to create long lasting quality and defined that each individuum has the responsibility to form his own but also help to form others mindset.

The second part showed us that Quality needs people who can communicate, take and give feedback without making it personal. To form a quality mindset means to understand that every person in a team has his own responsibility regarding quality, which only ends only if the project is terminated. Quality means also to have a clarity in visions, ideas, requirements, and expectation.

Today we'll talk about the different project types which we categories in two categories and about the technical implementation of quality.

The right choice: Project types

In the chapter "The truth about quality" we mentioned the following quote:

It's just a Proof of Concept but we would like to use it productive as soon as possible

We would answer that quote with something like that.

It would be better, if we think about the requirements and the expected outcome, maybe a PoC is not enough for your expected result.

Projects can be categorised into two major categories.

Project types to evaluate ideas

Quick, often fun, mostly timeboxed projects, where the developer tries a new technology or implements an idea quick and dirty. There is no need to think about long lasting sustainability or maintainability. With these projects we really try to proof that an idea or the use of certain technologies could work.

Proof of Concept (PoC)

As a rule of thumb, we go into a PoC with 8-12 person days, an it's most of the times one developer alone, who implements the PoC. Often there is no consideration of architectural concept no implementation of unit-test framework or the use of clean code principles. The focus for that kind of project is solely the proof that an idea could or could not work.

Prototype

The PoC worked, awesome! Let's implement a functional piece of software, which shows of the idea and might be pitched to as investor or stakeholder. Still we don't want to think too much about architecture or clean code or testing also we don't consider things like integration into existing systems or connecting database systems. Data which would be needed are mostly hardcoded or mocked. So, while prototyping the focus lies also time-based on the implementation of the idea in a presentable way, prototyping is mostly done in teams of 2-3 people.

Pilot

A pilot does have, like the PoC and the Prototype, a defined ending which is time-based to around 1-3 months. Pilots are usually done in teams. Architecture, patterns and the integration into existing systems are a big part of it. If a successful pilot is planned to go productive, then things like unit-test, versioning, CI/CD, coding-guidelines and clean-code patterns are implemented and considered as well so that the transition from pilot to product in productive environment is as smooth as possible.

Project types to implement an idea

Daily business for development teams is usually the implementation of new features in existing software. Within the project type for implanting ideas we differentiate between three kind of projects.

Legacy Project

Projects which are running in productive systems, where new features are implemented will be the main projects in the carrier of a software developer. Is the product long running, then a developer will work with an existing code base, where refactoring and embedding of legacy code into new features, written in new technologies, will be the main task.

Legacy Project's do have a high demand for mature quality mindset. Requirements or user-stories must be written very clearly and should consider all angles of the software, especially when the new feature will interact with the legacy part of the software.

Refactoring will be a big part of all legacy projects and can only be archived painless or with less pain, with a good test coverage (especially unit-tests and integration test) which will display regressions immediately. Therefor in most cases, the test coverage must be increased, before the actual refactoring can begin, which is time consuming, yes, but a necessity.

This kind of projects should have a very high automation throughout the whole software lifecycle, integration, deployment, delivery, quality and in productive environment monitoring. Continuous Quality should be implemented with corresponding quality gates which break the build and deploy pipeline if the quality demanded is not matched.

Since automation means no human interaction, a high automation guarantees a less errors in the different states of software lifecycle, which means higher quality.

Green Field Project

The unicorn in all the project types is a green field project. A project where the developer starts by clicking the "New Project" button in the IDE and make its first *git commit init*. At this point everything is new there are no technical debt no missing unit-test no legacy code which have to be embed by a weird workaround, just an empty project.

The development-team decide where it'll go from there, what language they are using, which frameworks will work best for the solution. They can evaluate best practises and technology stacks, without being pressured into an existing one. Green field projects can and should also start with a quality first approach.

Minimal Viable Product (MVP)

In an agile world we don't ship full developed software anymore, we ship features using DevOps and all the power of automation Continuous Integration, Deployment and Delivery gives us the ability to deliver multiple times a day to productive environment. An MVP is a new developed software either by a start-up (Green Field Project) or by an established company which shipped the first and most relevant business cases. A limited feature set to go to market as soon as possible, but with the promise of more features to come after the first version release.

The big advantage of working a project as an MVP is that the Business Unit can use the feedback of the end-user and build the an develop the software which is really needed. Quality wise, the same things are true as for Green Field Projects.

Implementation of a Quality first approach

Tools, tools, tools, our last and final chapter of this series will be about tools, that might help us secure the quality before we start even testing our code.

Environment

Modern software is complex, sure thing a developer develops his code on his local machine; however, it is not efficient to integrate, build and deploy everything on a local machine. Therefor we need some overall understanding about CI/CD (not necessary DevOps) about staging and containerization. Tools are a double-edged sword, if you learn them they could help you make you more efficient push your quality to a new level and help you keep it there, but if you use these tools with no knowledge or only rudimentary knowledge this tools will make everything worse and will add a complexity to your system which you can't handle.

Containerization

Not all to long ago a mediocre developer would answer QA in all seriousness that a bug is not there because "It work's on my machine". Using tools like Docker to containerize the software makes sure that everybody uses the exact same thing. Developer may develop into a container even when the develop local, the build job will create the exact same docker image as the developer uses for his development but with the new code base. Now the QA does have the exact same thing as the developer. Sounds perfect but it helps us only so far as people are willing to learn to use these tools. People tend to be lazy so the excuse "It works on my machine" evolved into "It works in my container".

Source Code Management

Since we work in teams, we need tools to collaborate with each other, Source Code Management tools like SVN, Mercurial or GIT are for sure the most important tools a developer can use. Since the feature set especially in GIT is huge, it is important that any person who works with code does know the tool very well. Also important for a successful collaboration is a strategy to manage the source code. There are a couple of good "best practise" branching models (e.g. Trunk-based, Git-Flow, ...). The development team should decide together what is the best approach for their workflow and keep to that practise.

Stages

To setup an environment means to display different states of the software lifecycle. Stages will help the team to determine, in which state the software is.

Development-Stage

That stage should be the playground for a developer, after the integration from the developer's local system, the development stage should be used as a first trial of the new implemented feature. It doesn't make sense to test UI- or any kind of Integration-Test since there will be all kind of non-stable features and configuration deployed.

Test-Stage

As the name suggests it, this is the stage where a stable and testable software will be deployed, this stage has similarities to the productive system, but not on a full scale. On this stage we use test-data, which are often synthesised, the access to third-party tools is often virtualized to save money, and since it is not relevant to test a third party-tool it is ok to get the expected answerer by a virtualization tool. Automated tests like System-Integration-, Ui-, Performance-, Load-Test will be executed on this stage, but also will this stage be used for manual-, explorative-, regressions- and security-tests.

Pre-Productive-Stage

In a perfect world the Pre-Productive-Stage would be a clone from the Productive-Stage. The new features are deployed here after they passed all the tests and are ready for productive. Business Unit now can do their acceptance tests and give the final ok to release the feature.

Productive-Stage

On this stage our code becomes the product, which hopefully, makes a lot of end users happy. To archive this, we need a multitude of monitoring system, which shows us the health of the environment as well as the product.

Code Quality

To ensure code quality we do have a huge variety of tools. We will not go into the tools but will point out what is important to consider - in our opinion - when thinking about code quality.

Coding Guidelines

Since we talked about communication in the most part of the series, and how to get to a level of understanding we need to continue this in the code we write. Coding guidelines are mandatory, there is just no way that any developer will write code however he sees fit. This will make the code unsearchable, unreadable and adds complexity into the maintenance. It's the common language we developer have and need to understand. Therefore, has each coding language it's predefined set of coding guidelines and are also integrated into their respective linters as well in static code-analysis tools like SonarQube, Fortify and co.

Clean Code & Solid principals

Is your code clean? Are your methods and class as short as possible but also as readable as possible and are they responsible for one thing only? Or do you have variables like

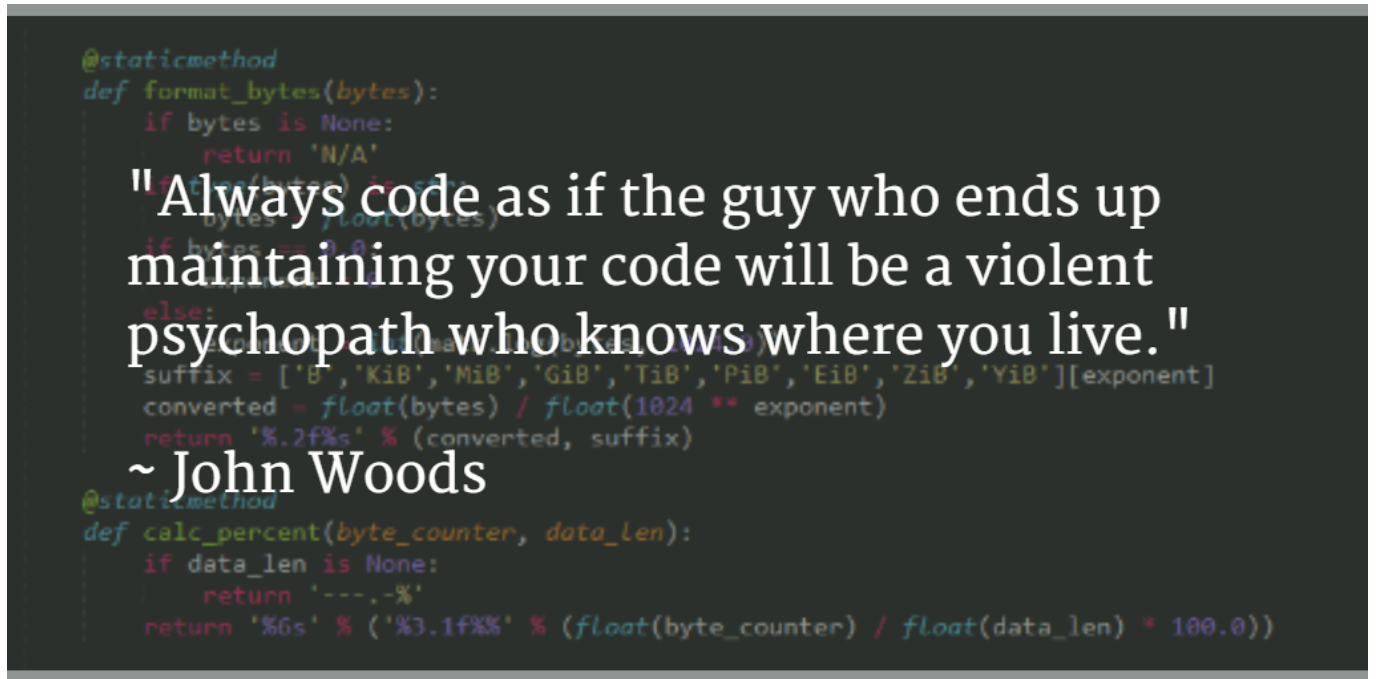
```
String a = "";
```

or methods with 400 lines of code where you have absolutely no clue what happens with an input and even less idea what the output of the method is?

To write code is your craftsmanship, don't treat this lightly. A chef will keep his knives always sharp and clean. A developer must do the same. The book Clean Code by Robert C. Martin must read, but more importantly has to understand and then use everybody who works with code. Get involved in meetups and discussion about clean code practises, your code should be reviewed, and you must take the feedback and learn from it. Clean Code and your mindset around it, has to evolve. As a rule of thumb, if you look at code you wrote 6 months ago and you wouldn't change a thing, then you are stuck in your personal development, which keeps you from being an awesome developer. Plan your implementation with the SOLID principles in mind:

- **S**ingle Responsibility Principle
- **O**pen – Close Principle
- **L**iskov substitution Principle
- **I**nterface segregation Principle
- **D**ependency inversion Principle

And finally, if you learned nothing from the series and will take nothing away from it, think about one thing:



"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

~ John Woods

Summary

To come to an end, developing software has a lot to offer and it's not all about code and configuration. To generate a good product requires a quality mindset on multiple levels. To say testing is our quality is wrong, and to late to begin with. Each step in a software lifecycle does need its quality assurance implemented in either a methodical or technical way.