Invisible Software Quality

14 November 2016 | **Insight Zuhlke** | **Arne Mertz**
Reading time: 5 minutes

When we think about what makes a piece of software good or even great, the terms that first come to mind usually are **functionality, usability, safety and reliability**. In other words, with good software it is easy to do everything it is designed for, and it is done well and without errors. Other important aspects are **performance and scalability**, i.e. the software works fast enough and for large enough data sets or large documents and with multiple users.

All these aspects are visible or even measurable aspects. The users of our software can tell if it looks ugly or runs slow or produces erroneous output. However, there are many aspects that are not visible to the user but can make some software better and more successful than others. A very important example is **software maintainability**.

Why does maintainability matter?

Software is not a static bunch of bits and bytes sitting on our hard drive that gets executed once in a while. Software has to be developed and evolved. Usually it starts off as a proof of concept or prototype, goes through alpha and beta testing until the first release. From there on features have to be added and changed and bugs need to be fixed. Good software caters to the needs of its users and adapts to new requirements that arise from the ever-changing world around it.

That means that the code at the heart of our software has to change constantly. The software has to be tested again and again to make sure that by changing the code we did not accidentally made it slower or break it by doing the wrong thing. Changing and adding code to fix bugs and add new functionality is called **software maintenance**.

Maintenance costs time and money – after all, someone has to do it. The less time and money it takes, the more successful the software can be. If we have to invest less in the development of a feature or in bug fixing, we can sell the software for less money or with even more and better features or a more awesome user interface.

How we can achieve software maintainability

In order to add or change code, software developers have to read and understand the code that is already there. Estimates about the reading to writing ratio vary wildly, but a common estimate is that **we spend about ten times more reading old code than writing new code**. That means that the main audience for the code we write should not be

the compiler or interpreter that produces the actual program, but the other developers that have to read what we are writing. As the saying goes, these “other developers” include especially our own future self, because we are very likely to come back to the code we have written.

Writing code for other developers means writing “clean code”, i.e. readable and understandable code. To achieve this, making it work the way we want it to work is only the first step. The next and sometimes harder step on the way to clean code is **refactoring**, i.e. to change it piece by piece, making it more readable while preserving the behavior of the software.

The key to successful refactoring are **automated tests**. A good test coverage can give developers the necessary confidence that the changes they apply to the code do not break existing behavior. The automated tests are preferably very fast so they can be run often to reduce the time code changes are left unchecked.

The cost of clean code and good tests

Writing tests takes time. Refactoring the code base also takes time. So the question arises whether the time we win by improving maintainability actually exceeds the time we have to invest into those actions that don’t appear to add visible value to our software. That question is a valid one and needs to be addressed.

Clean code practices require some initial time investment e.g. into setting up test frameworks. In addition, in the beginning projects are small enough so that the ratio of actually writing code to reading existing code is better, so the benefits of cleaner code are diminished. However, experience shows that over time, the benefits of refactoring and writing automated tests pay off.

How long it takes to reach the point of break-even depends on multiple factors, especially on the experience of the developers in using clean code techniques and writing tests. This experience does not only influence the time it takes to perform those tasks. Cleaning up code and achieving test coverage is not a yes/no decision, and the experienced **software craftsman** can assess how much code polishing is needed.

The bottom line is, that except for short term throwaway code like prototypes, a certain amount of refactoring and automated testing always pays off. And let’s be honest, that throwaway code tends to stick around longer than planned. Therefore, writing tests and refactoring code are not optional additions to the software development process. They are integral parts of it, like bug fixing, implementing features and writing documentation. All

these activities are needed to provide a software product of high quality, including the “invisible quality” software maintainability.