Factors Influencing Software Maintainability

dr. Adam Vanya

INTRODUCTION

In our previous blog post, titled “What Is Software Maintainability”, we introduced the concept of software maintainability and argued that it is important to control it. Carrying out the necessary control actions is only possible if we know which the influencing factors are. Describing these factors is the topic of this blog post. By monitoring and influencing those factors, one can control software maintainability.

THE INFLUENCING FACTORS

How easily a software system can be changed depends directly on two major factors: the state of the source code that needs to be changed and the developers executing the modifications. Let us look at the source code first. During the maintenance process, the source code is browsed and read by the developers to understand where and what should be changed. Most of the time, this means reading through source code written by other developers. The efficiency of this process is influenced by many source code properties. For instance, a well-structured software with small methods containing only a few decision points makes the process quick and efficient. On the other hand, a poorly structured monolith with huge methods having many decision points slows the process down enormously.

After analyzing the source code, introducing modifications is what developers do during the maintenance process. Also in this phase, the source code properties play a relevant role. For example, let us consider a situation where changing one duplicated code segment requires changing another one to carry out a consistent change. This extra work degrades the efficiency of the maintenance process. Also, such a situation could have been avoided by eliminating or not even introducing duplicates: duplicated code can often be extracted to methods which can then be called from multiple contexts. The mentioned situation causes further issues. If a developer forgets to change all the instances of the duplicated code segment then the inconsistency introduced will likely lead to errors or unwanted behavior. Fixing such errors requires extra maintenance time and effort spent.

When a developer believes to be ready with the implementation of a given functionality then it is time to test it. Proper testing covers all execution paths in the code so that all execution possibilities become checked against the requirements. Whether it is possible to test a piece of code properly and to which extent depends greatly on how the code is written. Decision points (like the ones introduced using if statements) in the code increase the amount of possible execution paths in the code to be checked. Methods with many decision points take a long time to test because of the amount of test cases which need to be implemented. Furthermore, there is a risk that only the most relevant test cases will be implemented because of time pressure and all the other execution paths will remain untested. This

introduces the risks of executing bugs during production time and causing availability issues or unexpected behavior.

Sometimes, due to how the code is implemented it is not even possible to test a piece of code as a unit. For instance, consider the an important portion of a very long method or an embedded SQL statement. To overcome these limitations, methods should be small and embedded SQL statements should be avoided when possible. Disregarding these guidelines can lead to untested code and also code duplicates with the risks already described.

The software maintenance activities mentioned above are further clarified in the 14764:2022 ISO/IEC standard. For this, please refer to [1]. As for the source code level properties (which also known as metrics), Zou and Kontogiannis in [2] provide a more systematic description.

Let us now look at the second major factor that influences maintainability: the developer. Giving the same development task on exactly the same codebase to a senior and a junior developer would most probably result in different execution times of the necessary code changes. That is because a senior developer has more professional knowledge and experience than the junior developer. Next to that, the senior developer is likely to alter the code in such a way that the altered code will be easier and faster to maintain for the next developer who will work on those altered code segments.

Independent from the level of professional knowledge, developers also have a certain level of cognitive capabilities. For instance, how many things can they remember at the same time. Such cognitive capabilities can influence how much time developers need to spend scrolling a long method up and down on the screen before the analyzed method is understood. For those developers, who can remember fewer things at the same time, it takes longer to analyze and modify the code.

Even if developers do have the necessary skills to create code which can easily be modified there are two further reasons why they would not do it and as a consequence why they would degrade maintainability. The first reason is the lack of motivation. Just as everyone, developers also need to be motivated to do things in a certain way. A self-respecting and experienced programmer typically has internal motivation to write maintainable source code. Other programmers may need to be motivated with sanctions and rewards to do the same. Sanctions can come, for instance, in a form of decreased bonus or even dismissal. Rewards could include advancing in positions with a salary increase and more respect.

The second reason is the lack of possibilities. In case a developer is skilled, experienced and motivated but is not provided the needed resources (for example, there is an extreme time pressure) then shortcuts will be applied. Since functionality is the number one priority in most development companies, this means developers will spend less time on maintainability. Even though this shortcut seems to be a solution in the short run, developers will spend more time because of the maintainability issues introduced in the long run. There is no escape, technical depth needs to be paid back with all the coupled interest. The concept of technical depth was coined by Ward Cunningham in [3] and it has received growing attention ever since.

SUMMARY AND CONCLUSIONS

One way or another, software maintainability is affecting you. It can have a significant impact on your career if you are working for a software development company and it is a major factor to consider when you invest in such companies or buy custom software. Therefore, we believe maintainability has to be understood and be controlled explicitly to reach your goals.

To help you get in control, we described and discussed the two major factors influencing software maintainability: the state of the source code and the developers. We argued that without properly crafted software and sufficiently skilled, cognitively capable, motivated and enabled developers it is not possible to reach an acceptable level of software maintainability.

The next blog post in our series on software maintainability will elaborate further on the potential effects of maintainability. With these blog posts, we hope to increase awareness about software maintainability, help you control this concept and start constructive discussions about it.

In case there are certain things described in this blog post you do not or not completely agree with or if you think it should be extended in some ways, please feel free to contact us and tell us about your idea: adam.vanya@frontendart.com

REFERENCES

[1] ISO/IEC 14764 (2022). ISO/IEC 14764:2022, Software Engineering — Software Life Cycle Processes — Maintenance

[2] Zou, Y., & Kontogiannis, K. (2002). Migration to object oriented platforms: a state transformation approach. International Conference on Software Maintenance, 2002. Proceedings., 530–539.

[3] Cunningham, W. (1992, December). The WyCash portfolio management system. In Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) (pp. 29-30).