Improving Software Maintainability

dr. Adam Vanya

INTRODUCTION

In our previous blog post titled “Sufficient Level of Software Maintainability”, we looked at the different approaches one can use to determine when the level of maintainability is sufficient. In case this level is not reached yet, actions need to be taken. Based on the results of the direct and indirect measurements of software maintainability, one may decide to invest in improving the level of maintainability. This blog post explains how it can be done.

IMPROVING SOFTWARE MAINTAINABILITY

One way to increase the maintainability level is to fix those maintainability relevant properties of the source code which hinder effective maintenance. We will go through some of the possible refactoring actions which can be performed to improve maintainability.

Splitting up long programming units (like methods or functions) into smaller ones can help ensure that the resulting units have a single responsibility and do only one thing. This refactoring step also increases the possibility to reuse existing code, it helps developers figuring out what a single unit does and it usually decreases the amount of decision points included in the units. Therefore, understanding and testing the resulting units may become easier as well.

Applying object-oriented principles (encapsulation, abstraction, inheritance and polymorphism) where needed can further increase maintainability. For instance, refactoring a piece code so that inheritance is used can decrease the number of decision points and execution flows to be understood and tested. Another possibility is to refactor programming units with many primitive parameters (like integers and strings) to use only a few object parameters instead. This requires explicit thinking about what a unit needs and creating the needed object representations based on the primitive parameters of the original unit. Reading, understanding and calling such refactored methods is easier, quicker and also safer.

Creating or refactoring high-level software components so that their number is manageable for the developers and in a way that their volume is approximately evenly distributed helps developers analyze the software system more efficiently and find what exactly should be changed or extended. Having a few equally sized components instead of a large component with several small components also decreases the chance of modifications propagating through the whole system and the effort needed to execute all the related tests.

Decreasing the ratio of redundant code caused by code duplications makes sure that developers will less likely introduce inconsistent changes into the code and leave bugs behind. Removing duplicated code also eliminates a significant amount of time needed to carefully think about where all the duplicates reside. In some cases, duplicated code fragments are located in multiple components. This means that a change to one of them may lead to change to the other one. By removing these types of duplications even some of the component coupling issues can be resolved.

Reducing the couplings between the components of the system (let it be static, logical, direct or indirect) lead to reduced development and testing efforts because less components may need to be analyzed, changed and tested. In large systems where multiple teams are working on the same system the mentioned refactoring may also reduce communication, organization and planning costs. Couplings between components are especially degrading maintainability when they form a cyclic loop: changing a component A starts a chain reaction of changing other components and as a consequence component A needs to be changed again. Such cyclic couplings bind the life-cycle of components together and makes maintenance complicated, time consuming and costly. Therefore, they have to be eliminated.

Keeping the implemented architecture consistent with the documented one ensures that the developers can trust and apply the information read from the architectural documentation without getting confused. Confused developers need time to figure out what they should follow and the time spent on clarification hinders effective development and therefore smooth maintenance.

Following coding related and technology specific best practices can enhance maintainability as well. For instance, empty catch blocks in Java try-catch type constructions may indicate that exceptions are not handled properly. Empty exception handling code makes it more difficult and time consuming to find the reasons for execution failures. There are many such source code related best practices that can be violated. Their identification and elimination is important to decrease their potential negative effect on maintainability.

Another major way to increase the maintainability level of a software system is to focus on the developers instead of the source code. As explained before, there are several developer specific concepts affecting maintainability. Based on these, we will now look at what can be done to improve software maintainability even further.

Coaching and educating developers to grow in their professional skills is an investment but it does pay off also from the viewpoint of maintainability. The more a developer knows about the technologies used to implement a software system, the development processes applied, the application domain and the concept of maintainability itself, the more efficient that developer will become not only to produce code but to produce it in such a way that is easier to maintain for the other developers as well. Of course, coaching and training sessions have to be effective. To be sure that they indeed are effective, proper measurements have to be planned, executed and evaluated before and after education. The results can also be used to form a good picture about the knowledge of a developer and choose the proper personal target together with the education needed to grow towards that target.

Next to improving technical skills, increasing the general cognitive capabilities of developers will also positively affect maintainability. A method in a Java class file might be relatively long, but if a developer is trained, for instance, to remember more things at the same time and how to find relevant information, then the need to scroll back and forth and to read the code of that method multiple times can be decreased. This helps developers to be more effective when changing or extending the source code.

Employing developers with already better skills and more experience is indeed more expensive due to an associated higher wage that needs to be paid. However, if we employ a developer whose technical and cognitive capabilities are measurably and significantly higher than those of a less performing developer then we will get more effective work and more tasks done right from the beginning with less education and training costs needed to reach the same level of knowledge.

Increasing the motivation of developers to create maintainable code is another prerequisite for maintainable source code to happen. This can be achieved, for instance, by introducing a reward and/or sanction system. Such a system can be based on end year bonuses, company public acknowledgements and the possibilities to climb up the company hierarchy, just to name a few possibilities. In any case, it has to be clear between the employer and the developer that writing maintainable code is a must to reach higher. Whether these goals are reached or not has to be continuously measured, discussed and rewarded (or sanctioned). Some measurements and also reviews can be embedded in the development process to prevent poorly maintainable code to be added to the codebase.

Providing the developers all the necessary assets (including sufficient amount of time) that they need to write maintainable code is frequently overlooked. This ignorance may happen, for instance, when we keep pushing developers to implement new features too quickly for a long time. Although we may have a good reason to do so, we have to be also aware that developers will rightfully apply maintainability related shortcuts (e.g. creating duplicates) and we can end up sacrificing maintainability for the hope of implementing more features. This strategy is, however, not sustainable in the long run. The amount of maintainability related issues will build up quickly and the more such issues are present, the longer it will take to implement new features. The result is a situation where the code needs to be refactored anyway before being able to implement the next new features. This can mean a long time interval when new features cannot be implemented at all. Therefore, it may be more suitable to address maintainability and allocate sufficient assets for it continuously than to do time consuming, and unplannable corrections every now and then the situation forces us to do so.

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 explicitly be controlled to reach your goals.

To help you improve maintainability, we discussed what type of relevant actions can be carried out. These actions are related mainly to the source code of a software system and the developers who need to maintain it. It turns out that there are many different types of actions that can be taken and for the best results these can be combined.

In case there are certain statements 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