Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living Software

Björn Latte
wobe-systems GmbH
Wittland 2-4
24109 Kiel, Germany
bl@wobe-systems.com

Sören Henning
Software Engineering Group
Kiel University
24098 Kiel, Germany
soeren.henning@email.uni-kiel.de

Maik Wojcieszak
wobe-systems GmbH
Wittland 2-4
24109 Kiel, Germany
mw@wobe-systems.com

***Abstract*—Maintaining a long-living software system is substantially related to the quality of the code the system is built from. In this experience report we describe how a set of practices and tools has been established and used on the early stages of a project. The approach is based on Clean Code and the use of well known static code analysis tools. The tools and practices have been used with an immediate effect of having cleaner code that is easier to understand in the long term. Additional attention is given to the cultural aspect that is involved in reaching a mindset that will allow to set and uphold code quality standards. Reaching a common understanding is a team effort that requires ”leaving one's comfort zone“. Finding common ground can significantly decide about failure or success in creating maintainable code.**

I. Introduction

Building long-living software involves many decisions that influence maintainability and extensibility of a system throughout its life-cycle. In early stages of building a software-system the focus is often on a certain set of platform technologies and state of the art patterns. During its following lifetime the software-system will be challenged by inevitable change of requirements, increased workload triggering performance issues, modernization requests due to emerging technologies and the like.

Evolution and extension of a long-living software often requires in-depth knowledge about the system which is only present with senior engineers who have been working on the system for a long time. Therefore, teams often have a *team historian* whose knowledge is used whenever a source code related question is raised [1]. In many cases this *code knowledge* is also necessary due to the state of the code itself.

With the focus mostly being put on technology and patterns in the early stage of a project, the code itself is given less attention. While allowing rapid movement and aggressive changes in the beginning, this will turn up later as hard to understand, maintain, and enhance code – effectively increasing the time and cost needed to accomplish the necessary evolution. This effect is commonly referred to as technical dept, a term coined by Cunningham [2]. Additionally, with code that is hard to understand the potential of a *bad fix* in case of a bug increases. Also it will become a problem to provide proper tests with in-depth coverage [3]. Even more so when tests have to be added at a later stage.

This research is funded by the Federal Ministry of Education and Research (BMBF, Germany) in the Titan project (https://www.industrial-devops.org, contract no. 01IS17084A–B).

There are plenty of good papers available on tools to analyze code for its general quality and also to alert to potential defects, for example [4]. What we observe though is that often enough little use is made of the available means to produce good quality code from the very early stages of a software project. The same can be observed during the education of future software engineers, which focuses on algorithms and architecture but seldom on code quality [5].

Through the introduction of agile software development and a cultural move towards DevOps, a versatile set of tool-chains and practices has been established for rapid and constant evolution of long-living software. We believe that combining this with clean code [6] and enforcement of quality via build pipelines and review processes will lead to more maintainable software systems.

We present an experience report on how achieving high quality code has been implemented in the early stages of the project “Titan – Industrial DevOps platform for iterative process integration and automation” (Titan) [7]. The project's goal is to provide a long-living distributed software platform for high volume data processing. Flow-Based Programming (FBP) [8] will enable the end-users to model their data processing graphically without having to write code.

II. Project Outline

The Titan project aims to explore and build the prototype of a scalable, resilient, and distributed software platform for small and medium enterprises (SME). Additionally, by transferring the principles of DevOps into the industrial domain, the platform is sided by the methods of Industrial DevOps. This will enable SMEs to use agile processes for a larger number of roles present even in small organizations. The project is executed as a cooperation between a software house and a university. The original proposal for the project already included the use of lean and agile methodology during the research and implementation phase. Also a focus was turned towards code quality and longevity of the platform.

The Titan platform itself will be made available as open source software. Open sourcing is aimed at reducing the *vendor lock-in* that comes with most other software platforms used by SMEs. On the other hand, open source software is tarnished by the perception of being less secure and of lower quality then commercial software. While the factor of quality and security cannot be validated for commercial software without gaining access to source code this insight is possible for open source.

III. Approach

From the very start of the Titan project an approach has been taken to enforce a route to produce readable and maintainable code. This approach is based on four factors which are described in the following.

A. Clean Code and Code Review

The code used to build a software system is often written by software engineers which come from several backgrounds and have varying experience. Different parts of a system may use the same programming language or, as it is the case for the Titan project, multiple languages: 1) core libraries use C, 2) prototype backend processes Python or Java, 3) frontend applications JavaScript, TypeScript, and ECMAScript6. Open source software allows complete access to the code to understand or scrutinize the inner workings of the software. On the other hand it heavily relies on contributions from the outside to extend and modify it. Attracting new contributors thus is highly related to the code being understandable not only to its original author. Over time many bad experiences of developers with legacy code have led to the "Clean Code" movement which has put forth a set of practices and principles that ease the writing of good code.

Applying and enforcing these practices from the very start becomes a key point in avoiding *technical dept* which is hard to eliminate later on. Besides educating individual developers about the practices to use, our approach is sided by several other aspects. One of them is selecting or authoring "code style guides" appropriate to the programming language of a (sub)project. The guides are reviewed and agreed on by the core team of the project. Enforcing style is not limited to adhering to a fixed style of 1) code formatting, but also in 2) consistent use of language features e.g.: using or omitting the "this" statement in Java code, 3) naming conventions, and 4) use of meaningful names. The first two items can easily be checked and enforced in an automated way using static code analysis. This can happen inside an Integrated Development Environment (IDE) or in build pipelines of Continuous Integration environments. The latter two are only partially covered by analysis tools and are best addressed by making use of code review processes. Code that is committed to any of the repositories of the Titan project is cross-checked by a second developer before being accepted into the master branch. If *code smells* or non compliance are identified, the original developer and the reviewer enter into a discussion and resolve the problem by immediate refactoring. This requires a large amount of openness and trust between the parties involved in the discussion. The same openness and trust is also required throughout the whole project team. We experienced this to be a learning process with a rather steep learning curve and expect this to be likewise for most teams. The most prominent issue is overcoming the psychological aspect involved on constructive criticism and positive feedback.

B. Test-Driven Development (TDD)

Using Test Driven Development, where applicable, from the start forces developers into certain habits while coding. At the same time TDD lays the ground for later refactoring. Being forced to first write a test and then only write as much code as to make it pass the test will keep developers working in a very short loop. Classes and functions will be kept short and focused on the single task that they are intended to perform again delivering cleaner code. The availability of a test can be used straight away to refactor the very function that is currently being worked on, helping the developer in making changes without breaking the functionality.

In the long term the availability of comprehensive test coverage will enable developers to perform larger refactoring [9] of the code. Lacking tests, changes that break functionality can be introduced without even noticing. Maintainability diminishes and the potential for introducing false fixes rises significantly.

C. Static Code Analysis

Static analysis tools analyze the source code with regard to several aspects ranging from syntax checks to detecting coding mistakes and security vulnerabilities. Depending on the programming language, different sets of tools exist for static analysis. While many tools only target a single aspect of code quality, other cover multiple aspects. With the latter some overlap in their functionality can be found.

Even though tools that do static analysis of source code have been around for a long time they are seldom used with smaller or prototype projects [10]. Our approach is to include these tools from the early stages on. Analysis comes in several forms from syntax checks to detecting coding mistakes and security vulnerabilities. Depending on the programming language used for individual subprojects of Titan a combination of analysis tools to cover multiple aspects is employed. C code is checked by splint[1] and flawfinder[2] to detect coding mistakes and known insecure coding patterns. Additionally, GNU complexity[3] is used on C code to measure and minimize the code's complexity with readability and maintainability in mind. Subprojects using Python are checked with pylint[4] and flake8[5] to enforce code style and formatting as well as

[1] http://splint.org

[2] https://dwheeler.com/flawfinder

[3] https://www.gnu.org/software/complexity

[4] https://www.pylint.org

[5] https://gitlab.com/pycqa/flake8

calculating code complexity and detecting code smells. In Java-based projects we use a combination of Checkstyle[6], PMD[7], and SpotBugs[8] to detect code style, formatting, and partially logical as well as performance and security issues.

We execute these static analysis tools during different phases of the software development process. Firstly, the software developer uses them while writing code. In addition to a manual, regular execution, we integrate the tools into the Integrated Development Environment if this is supported. Violations of coding guidelines or potential bugs are automatically highlighted in the source code. Secondly, the static analysis tools are executed during the local build process, for example, via build tools such as Gradle[9]. Both approaches together have been proven beneficial as they directly provide feedback on the quality of code and also serve as a guidance factor. However, making use of them can not strictly be enforced in this way. In order to ensure a certain degree of code quality in the common code base, defined quality gates have to be checked in a compulsory, automated, and uniform way as described in the following.

D. Continuous Integration/Continuous Delivery (CI/CD)

Automating tasks that are executed repeatedly and manually is one of the main objectives when employing DevOps principles. Directly accompanying the DevOps focus of the Titan project by making use of automation from the very beginning for its build environment is the logical choice in this case. CI also becomes the main step in enforcing code quality measures. In our approach the former mentioned static code analysis is performed as a build step in each of the subprojects build pipelines. Analysis is performed with the tool or tools matching the programming language of the subproject. Configurations for the static analysis tools are made part of the source code of the project. This is done in the form of configuration files for the static analyzers or directly in the steps of the declarative pipeline script.

Generally, using declarative pipeline job descriptions that are handled as checked-in code does enhance the transparency. Developers can change, add to, and adjust the build steps by making changes to the declaration. The configuration is not hidden in a separate system providing the build infrastructure. Through using GitLab[10] as our choice for code hosting and CI/CD, building and running pipelines becomes a self-service. Developers gain direct access by adding the declarative pipeline script to their project without further need for configuration in a separate interface of a CI/CD environment like Jenkins. Build pipelines that generate artifacts and deliver feedback to the developer are immediately triggered on the next push of code to the hosting platform.

Pipeline steps that provide quality gates should be set to fail, if any of the analysis tools reports flaws or metrics that exceed the thresholds defined, effectively forcing the developer to refactor the code right away. With the next improvement cycle the pipeline possibly can be completed again and artifacts that meet the set-out standards become available for deployment.

IV. Initial Results

Using the approach described above, we observed immediate improvement on code quality and use of secure coding standards. Even though the Titan project is still in its early stages this already became evident not only due to the metrics used, but also during the manual reviews. First commits of code did seldom meet initially set quality standards. This observation became especially apparent when first running GNU complexity on the C code subprojects. During the first iterations of Titan subproject ujotypes-c[11], for instance, 21 functions were reported being above a complexity level of 5, the set threshold. The number of functions was lowered to 8 within a few iterations. The overall maximum complexity was lowered from 37 to 13 for the most complex function. With a complexity value of 37 being considered as: “Difficult to maintain code”, whereas 13 is to be: “Maintained with little trouble” [11]. The observed complexities also lead to the result that a certain amount of complexity is unavoidable and also acceptable. This threshold of acceptable complexity needs to be agreed on in the project’s team and needs to be properly documented for future reference. We found measures to lower complexity in an iterative process of learning and communication. This required a *culture of openness* that first had to be established within the project group. In many instances this included “leaving ones own comfort zone” and to adjust a behavior, improve a skill, or learn something new. For some project members this is harder to achieve than for others, even if all share the common goal of creating a long-living software system. Setting goals, achieving common understanding, and defining code quality standards is a team effort that includes a lot of controversy. Finding common ground and committing to a common understanding can significantly decide about failure or success in creating maintainable code. With the intention of open sourcing the project, the same level of openness and common understanding will have to be achieved in a much larger community.

In our Java-based subprojects[12], static analysis tools were not introduced until after an initial prototyping phase. In this context, we also decided to apply established code formatting guidelines. Using the tool Checkstyle forced us to comply with the code style guidelines as all violations were reported as errors. Besides formatting rules, PMD and Checkstyle identified, for example, for the *Control Center History* subproject [12] 66 issues of other origin. Most of them can be assigned to categories such as *Error Prone*, *Code Style*, or *Design*. SpotBugs, which also targets logical bugs and exploitable security vulnerabilities, did not find any additional issues. With integrating the tools as automatic checks into the CI/CD

[6]http://checkstyle.sourceforge.net
[7]https://pmd.github.io
[8]https://spotbugs.github.io
[9]https://gradle.org
[10]http://gitlab.org

[11]https://git.industrial-devops.org/titan/related-projects/ujotypes-py
[12]https://github.com/cau-se/titan-ccp

pipelines, all initially detected 1716 issue were addressed and the code refactored accordingly. We conclude that using static analysis tools enormously supports producing clean code. Moreover, we were able to notice that without these tools several aspects of clean code are easily left unconsidered. This primarily affected aspects of readability, comprehensibility by others, and maintainability. These aspects come from areas, where developers assume that their personal style will also appeal to others. On top of this the code analysis revealed performance issues and potential vulnerabilities.

A main drawback in our opinion can be found in the usability of the feedback provided by the static analysis tools and the visibility of results in the CI/CD environment. The static analysis tools reported flaws, insecure coding or exceeded complexity but mostly gave little guidance as to how to counter the reported issue. The lack of guidance leads to a situation where individual developers repeatedly, and individually go on a search for a solution to the same problem potentially with a varying outcome. The same observation has been made by Johnson *et al.* [10] in the paper on: "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?". The second aspect of lacking visibility is only partly addressed by our chosen toolchain. While code review processes are enhanced by the features provided by the GitLab code hosting platform the results of static analysis tools stay hidden in the job logs of build pipelines. A more visible and guiding representation of results would be beneficial to the developers which coincides with an observation made by Sadowski *et al.* [13]: "For a static analysis project to succeed developers must feel they benefit from and enjoy using it".

V. Conclusion

Based on our experiences during the early stages of the Titan project we presented an approach that improved source code quality from the beginning of software development on. Our goal of reaching clean code and essential maintainability in a long-living software platform is substantially supported. The project did not yet experience outside evolutionary pressure besides the internal iterative aspect of agile processes, because of this the chosen approach requires further validation throughout the project. None the less we expect using a set like the presented to help avoid degradation of maintainability. The choice of procedures and practices is more a cultural than a technical issue. There are plenty of tooling solutions available that have a long history and proven background. Making best use of them though requires a mindset of openness between developers to create acceptance. A common understanding has to be reached as to why code needs to adhere to committed quality standards. The DevOps culture and the agile software development practices provide a fruitful ground for employing the described approach.

Not only clean code aspects can degrade over time, but also quantitative quality characteristics such as performance have to be maintained for long-living software. As future work, we therefore plan to integrate performance benchmarks into the CI/CD infrastructure as suggested by Waller *et al.* [14].

References

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: ACM, 2006, pp. 492–501.

[2] W. Cunningham, "The WyCash Portfolio Management System," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992.

[3] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[4] P. Tomas, M. Escalona, and M. Mejias, "Open source tools for measuring the internal quality of java software products. a survey," *Computer Standards & Interfaces*, vol. 36, no. 1, pp. 244–255, Nov. 2013.

[5] D. Schmedding, A. Vasileva, and J. Remmers, "Clean Code – ein neues Ziel im Software-Praktikum.," in *Tagungsband des 14. Workshops "Software Engineering im Unterricht der Hochschulen" 2015*, (Dresden, Germany), ser. CEUR Workshop Proceedings, Aachen, 2015, pp. 81–91. [Online]. Available: http://ceur-ws.org/Vol-1332/paper_10.pdf.

[6] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[7] Titan Project, *The industrial DevOps platform for agile process integration and automatisation*, Accessed: 2018-12-19, 2018. [Online]. Available: https://industrial-devops.org.

[8] J. P. Morrison, *Flow-Based Programming, 2nd ed.: A New Approach to Application Development*. Paramount, CA: CreateSpace, 2010.

[9] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 672–681.

[11] Free Software Foundation, Inc., *GNU complexity manual*, Accessed: 2018-12-19, 2011. [Online]. Available: https://www.gnu.org/software/complexity/manual/.

[12] S. Henning, W. Hasselbring, and A. Möbius, "A scalable architecture for power consumption monitoring in industrial production environments," in *IEEE International Conference on Fog Computing*, in press, 2019.

[13] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[14] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including performance benchmarks into continuous integration to enable DevOps," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 2, pp. 1–4, Mar. 2015.