embeddedworld2015
Exhibition&Conference
... it´s a smarter world

Managing Product Quality in Complex Software Development Projects

Experiences Gained at Audi

Kai Schüler, Ralf Trogus
AUDI AG
IT Elektrik/Elektronik
Ingolstadt, Germany
kai.schueler@audi.de, ralf.trogus@audi.de

Martin Feilkas, Thomas Kinnen
CQSE GmbH
Garching b. München, Germany
feilkas@cqse.eu, kinnen@cqse.eu

***Abstract*—Software systems are often maintained by different development teams during their lifetime. Knowledge transfer is a very critical point when a system is passed from one team to another. The source code's quality is a key to successful sourcing of software development activities. However, if no measures are enforced, the code's quality continuously decays over time. In order to avoid this decay and to even improve the quality of grown software systems, an additional quality control process has been established for a complex software engineering project at Audi. This paper describes the new quality process and presents the gained experiences.**

Keywords—software quality management; software quality analysis; quality control; software development outsourcing

I. INTRODUCTION

Software decays over time: It becomes more and more incomprehensible, its architecture erodes and accidental complexity rises. These facts on long term maintainability have been reported by many studies in the last 30 years of software engineering research [1][2]. The situation becomes even worse if there is turnover in the development team maintaining the code. Nevertheless, in today's software engineering projects, developer fluctuation is inevitable during the life-cycle of any larger system. Companies buying software development services need a certain degree of independence from specific suppliers or even individual developers without losing the capability to further evolve the code base with new personnel.

A sufficient level of comprehensibility and maintainability (internal software quality), which is preserved over the software's evolution, is the key to manage hand-overs from one supplier to another. However, internal software quality is often an undisclosed property in software engineering projects. To shed light onto the status of internal quality, different quality aspects need to be measured and actively controlled, such as high maintainability/comprehensibility, architecture preservation, and test quality.

This paper presents experiences gained from introducing the new quality control process and corresponding analysis tools in a complex software development project at Audi, in which five different software suppliers are actively involved. Carefully selected metrics were used as key performance indicators (KPIs) to assess the software quality on a regular basis. This information was used for making the internal software quality transparent for management. Additionally, explicit tasks for quality improvement have been filed to address and schedule software quality issues within the development process. In this paper the technical and social challenges will be discussed. Furthermore, quantitative measures and trends of the selected KPIs will be presented that demonstrate the positive effect of this process on the system's quality.

II. STARTING POINT AND REQUIREMENTS

The project that was chosen for the evaluation of the quality control process has been in development for about 10 years and contains about 570,000 LoC of Java and 210,000 LoC of Python. Currently, about five suppliers are involved in the project, with some fluctuation in the past. The system is in broad practical use at Audi and at many subsidiary companies of the Volkswagen AG such as Porsche, Lamborghini, Seat and Skoda.

There were many requirements that had to be fulfilled due to the general set-up of the project. The most important ones were the following:

R1: Audi has a positive and cooperative relationship to its suppliers. The introduction of the quality control process should not be regarded as paternalism, but as a service for the development team.

R2: Keep quality improvement cost-effective: Quality is no end-in-itself. Quality must be made transparent; however, the spending on quality improvement must match with the system's lifecycle.

R3: Low risk of new bugs due to improvements of internal quality: Every change performed in a codebase comes with a risk of introducing new bugs, which must be avoided.

R4: In many development projects, the amount of external developers outnumbers internal staff by far, thus the control process must be efficient. This must be achieved using a high degree of automation.

III. Measuring Software Quality

Many metrics have been proposed claiming to measure certain aspects of software quality but did not prove to be effective in practice (e.g. [4][8]). In practice a quality control process will only be successful if the quality metrics are accepted by all roles in the process. This is an important precondition for R1. To avoid typical pitfalls of software quality measurement, every metric used must fulfill the following criteria:

- *Objective*: The way the metric is calculated on the code must be clear to every developer. Also the way of aggregation, e.g. from a set of measurements on method-level to one number for the complete codebase, must be transparent. A counterexample for this criterion would be the maintainability index [8], which is based on a very complex formula that is not easily comprehensible.
- *Implications of code changes to the metric must be clear*: It must be predictable how the metric is affected by changes to the code. It is very important that an optimization of the metric cannot be achieved by making the code quality worse (e.g. a metric counting method length is improved by removing comments).
- *Actionable*: If a metric results in a non-optimal value, the actions to improve the situation must be clearly deducible.
- *Clear impact on development/maintenance activities*: In order to gain acceptance for the metric, its impact on development activities, like code reading, testing etc. must be clear – it must be clear that a bad metric value really indicates more time consuming or error-prone development.

The first step to define the quality metrics was to inspect existing guidelines and have a critical look at them. We found several metrics that were already proposed in the documents. However, in some cases we decided not to use them as they did not meet the above mentioned criteria. For example, cyclomatic complexity, although frequently used in practice, does not reflect the complexity as perceived by developers very well [4][5].

To fulfill R4, all rules in available coding guidelines, which were automatically checkable, have been implemented in the tool Teamscale [3] using its custom check framework. Thus, the efforts for manually checking certain quality aspects can be reduced (R4).

The following metrics have been chosen as KPIs in the control process:

KPI	Measurement	Thresholds
File Size	Files up to 300 SLOC [1] are rated as green. Files between 300 and 750 SLOC are considered as long and thus rated as yellow. Files longer than 750 SLOC are considered as very long and thus rated as red.	Less than 5% of the code rated as red and less than 25% rated as yellow or red.
Method Length	Methods up to 30 SLOC are rated as green. Methods above 30 SLOC and up to 75 SLOC are rated as yellow. Methods longer than 75 SLOC are rated as red.	Less than 5% of the code in methods rated as red and less than 25% rated as yellow or red.
Nesting Depth	The nesting depth of the source code is measured. Code regions with a maximum nesting depth up to 4 are rated as green. Regions with a maximum nesting of 5 are rated as yellow. Regions with a nesting of 6 or above are rated as red.	Less than 5% of the code in methods rated as red and less than 25% rated as yellow or red.
Code Duplication	To quantify the extent of code duplication, the "clone coverage" metric is used. This number can be interpreted as the probability that a randomly selected statement of the system has been copied at least once and thus a change has to be propagated to other parts of the system.	Clone coverage lower than 10%.
Code Anomalies	A multitude of checks are performed, e.g. adherence to coding guidelines or usage of error-prone constructs.	Not more than 10 anomalies per 1000 LOC.
Code Comments	All public types, classes and methods must be commented. Trivial Getters/Setters and overridden methods are excluded.	No missing comments.
Test Coverage	Statement coverage is measured for all automated unit tests.	Component-specific thresholds for test coverage.
Architecture Conformance	Dependencies from the code are extracted and checked against a machine-readable model of the intended architecture.	No architecture violations.

IV. The Quality Control Process

The following section explains the roles and responsibilities in the quality control process. After that, the application of the quality process on a grown codebase is introduced.

A. Roles and Responsibilities

One of the most critical factors in successful quality control is the point in time the quality checks are performed. If quality is only measured when releases are shipped, there is usually no time left to actually improve quality. Therefore, a continuous quality control process, which is performed as a flanking activity to software development, was applied.

Fig. 1 presents the established process containing two control-loops. The first one is achieved by directly giving every developer feedback to the changes he performed. Every commit is analyzed by the tool Teamscale. Due to the unique incremental analysis approach of Teamscale, developers receive an immediate reaction after every commit, informing them about quality deficits that they added or removed with their changes. Also deficits that are not new but reside in the context of modified code (e.g. in the same method) are reported

[1] SLOC (**S**ource **L**ines **O**f **C**ode): lines of code without comments and empty lines

as missed improvement chances. This control loop is very efficient because the developers are still familiar with the task they were working on, thus no additional familiarization and program comprehension efforts need to be invested.

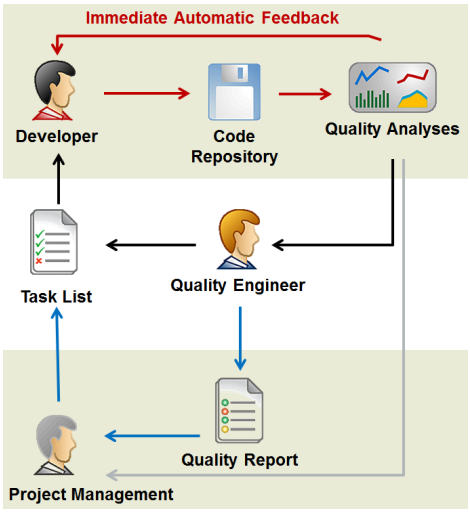

Fig. 1. The Quality Control Process

The second control loop is driven in a lower frequency, e.g. every 3-5 weeks, by a role called the quality engineer. The quality engineer is responsible for writing quality reports to inform the project management about the quality status using metrics and trends as well as writing lists of tasks for improvement of the software quality. The tasks and quality metrics are usually contained in a single document.

The overall quality status in the quality report is presented as an assessment based on traffic light ratings. The trends reflect the changes of the individual metrics since the last report. For creating the task list, the quality engineer examines all deficits that represent violations of the quality goal since the last report. In some cases, a prioritization is needed according to the severity of the deficits. The tasks must be written to contain a concrete proposal of how the deficit should be removed. Thus, tasks must not be written in a way "remove clone X" but always provide a feasible solution "remove clone X, by introducing a method Y and moving it into base class Z". On the one hand, this forces the quality engineer to carefully think about the findings produced by the analysis tools and on the other hand, it has an educational effect on the developers. Besides writing new tasks, the quality engineer also inspects the tasks that have been closed since the last report and checks if the chosen solution is sufficient. This activity prevents an optimization against metrics without actually improving the system's quality.

In software engineering projects, there is usually competition between the implementation of features versus quality improvement tasks. However, project managers are most aware of the lists of features to be implemented whereas quality aspects usually remain hidden. By making quality transparent by reporting continuously on the quality status and trend, project managers can deal with the feature vs. quality competition in a more informed way.

B. Grown Software

Introducing quality metrics and software quality analyses right at the beginning of a (green field) software engineering project is easy because a strict zero findings policy can be followed from the first line of code. However, the goal was applying the process to systems that have been in practical use for years. Additionally, quality control was introduced in a running project while many critical functional changes were performed on the system by the developers.

Although the software quality of the examined system was quite good, the source code's first analysis revealed many deviations from the existing coding guidelines on the one hand and (of course) the additionally introduced quality analyses on the other. Cleaning up the complete codebase was not an option, due to unrealistic efforts that would have been needed with high risks of introducing new bugs. This would contradict our requirements R2 and R3. Thus, we had to accept the situation as it was and define a method to gradually improve the situation.

Fig. 2 shows a hierarchy of so-called quality goals. The quality goal "perfect" was not achievable under realistic conditions and "indifferent" did not match the requirement that the system should be further maintained and extended for decades. Thus, a middle course was needed. The quality goal "preserving" would mean to accept all deficits in the code but not allowing any new defects to be introduced, while "improving" would even force developers to clean-up old code when they modify it. Consequently, the quality goal "improving" seemed to be the right choice for this system, because it matched the system's lifecycle and Audi's future plans best.

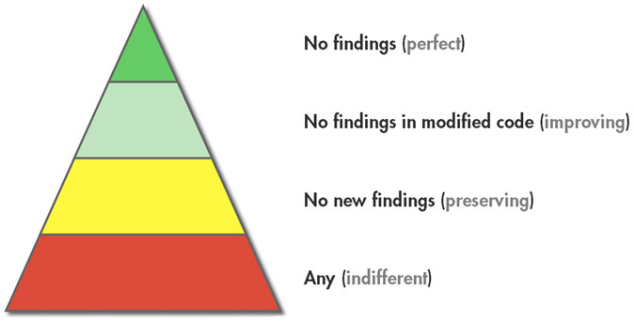

Fig. 2. A Hierarchy of Quality Goals

This decision was appreciated by the developers as it meant not to proactively modify the very old and stable parts of the code. This lowered the risk of introducing additional bugs (R3). Furthermore, developers only had to clean-up code they were changing and testing anyway. Thus, no additional efforts for program comprehension and testing are needed (R2).

In order to check the quality goals "preserving" and "improving", analysis tools must be capable for performing a baseline-delta analysis. Therefore, they have to be aware of the birth of every finding to filter the old ones. As the analyses in Teamscale are driven by individual commits, Teamscale is able to use the information from the version control system to make

a clear distinction between old and new findings. Even if files are renamed or code is moved between files, Teamscale is still capable of tracking the findings as they move with the code [6]. This precision is very important to not end up with having all findings regarded as new after small refactorings.

V. RESULTS

The most important result of the project was the broad appreciation from the developers for the introduction of the quality control process. Although there were critical voices at first, the selection of the KPIs, based on the criteria described above, and the application of the quality goal "improving", not forcing the developers to proactively clean-up old and stable code, lead to a broad consensus in the team. The fact that also the project managers were aware of the quality-evolution also had a very positive effect for the developers because it was now easier for them to argue when more time was needed to achieve the quality they desired.

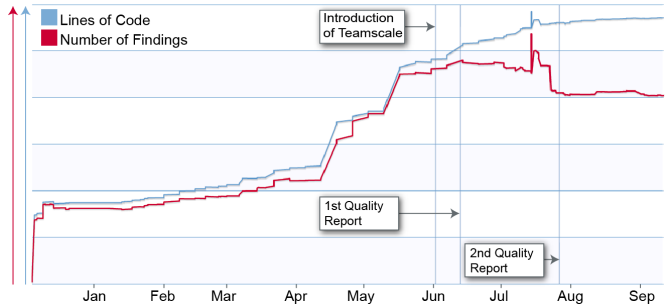

Fig. 3. Lines of Code and Findings Trends

Another social aspect was crucial for achieving R1: The role of the quality engineer was handled in a way that the developers regarded his work as a service for them and the project. The role must not have the appeal of a "quality police" hunting for "quality felons". This enabled a very constructive dialog on how to achieve the best solutions between the developers and architects with the quality engineer.

Quality Indicator	Trend	
File Size	Less code in long files (-2.3%)	
Method Length	Less code in long methods (-2.1%)	
Nesting Depth	Less deeply nested code (-1.4%)	
Clone Coverage	Less clone coverage (-1.4%)	
Code Anomalies	Less anomalies per 1000 lines (-1.6 per 1000 lines)	
Comments	Less missing interface comments (-154; -0.3 per 1000 lines)	
Architecture Conformance	Less violations (-9)	

Fig. 4. Management Summary of the second Quality Report

The success of the quality control process could be observed very early in the project. Fig. 3 shows the trend of the overall deficits that Teamscale detects on the code (red line) and the continued growth of the system in lines of code (blue line). After the first quality report the number of deficits already drops although the system is still growing (a similar effect was already reported in the studies performed in [7]). This positive trend was another motivating factor for the whole development team.

According to statements of the developers, the Teamscale IDE integration, providing markers for deficits without having to execute the analyses on their local computers as well as the early (within seconds) and personal feedback after every commit, was a crucial success factor. This positive trend continued in the second quality report. As Fig. 4 shows all quality indicators were improved during the reporting interval. The second quality report displays the code quality's trend during the last iteration as it will take several months if not years to significantly change the overall assessments due to the codebase's size.

VI. CONCLUSIONS AND FUTURE WORK

The key success factors for introducing the quality control process in the project at Audi were:

- A carefully selected set of quality KPIs with a broad acceptance in the development team.
- Usage of the quality goal "improving" where developers clean-up code only when they modify it (or create new code), not forcing proactive clean-up in old and stable components.
- High degree of automatic checking aligned with immediate and personal feedback to developers after every commit and directly in the IDE using Teamscale.

Due to the success and the positive reputation of the project at Audi, several other project coordinators already reported interest in introducing a similar process. Thus, the plan is to expand the usage of this method to other software development activities at Audi in order to establish it as a broadly used practice.

REFERENCES

[1] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution – the nineties view. In Proc. of the International Symposium on Software Metrics. IEEE CS Press, 1997

[2] D. L. Parnas. Software aging. In Proc. of the International Conference on Software Engineering (ICSE), pages 279–287. IEEE CS Press, 1994.

[3] L. Heinemann, B. Hummel, D. Steidl. Teamscale: Software Quality Control in Real-Time. Proc. of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14), 2014.

[4] B. Curtis et al. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics, IEEE Transactions on Software Engineering, Volume 5, Issue 2, 1979.

[5] B. Katzmarski, R. Koschke: Program complexity metrics and programmer opinions, IEEE 20th International Conference on Program Comprehension (ICPC), ISSN 1092-8138, 2012.

[6] D. Steidl, B. Hummel, E. Juergens. Incremental Origin Analysis of Source Code Files. Proc. of the 11th Working Conference on Mining Software Repositories (MSR'14), 2014.

[7] D. Steidl et al. Continuous Software Quality Control in Practice. 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14), 2014.

[8] D. Ash, J. Alderete, P. W. Oman, and B. Lowther. Using software maintainability models to track code health. In Proc. of the International Conference on Software Maintenance (ICSM), pages 154–160. IEEE CS Press, 1994.