Continuous Software Quality Control in Practice

Daniela Steidl*, Florian Deissenboeck*, Martin Poehlmann*, Robert Heinke†, Bärbel Uhink-Mergenthaler†
* CQSE GmbH, Garching b. München, Germany
† Munich RE, München, Germany

***Abstract*—Many companies struggle with unexpectedly high maintenance costs for their software development which are often caused by insufficient code quality. Although companies often use static analyses tools, they do not derive consequences from the metric results and, hence, the code quality does not actually improve. We provide an experience report of the quality consulting company CQSE, and show how code quality can be improved in practice: we revise our former expectations on quality control from [1] and propose an enhanced continuous quality control process which requires the combination of metrics, manual action, and a close cooperation between quality engineers, developers, and managers. We show the applicability of our approach with a case study on 41 systems of Munich RE and demonstrate its impact.**

I. Introduction

Software systems evolve over time and are often maintained for decades. Without effective counter measures, the quality of software systems gradually decays [2], [3] and maintenance costs increase. To avoid quality decay, *continuous quality control* is necessary during development and later maintenance [1]: for us, quality control comprises all activities to monitor the system's current quality status and to ensure that the quality meets the quality goal (defined by the principal who outsourced the software development or the development team itself).

Research has proposed various metrics to assess software quality, including structural metrics[1] or code duplication, and has led to a massive development of analysis tools [4]. Much of current research focuses on better metrics and better tools [1], and mature tools such as ConQAT [5], Teamscale [6], or Sonar[2] have been available for several years.

In [1], we briefly illustrated how tools should be combined with manual reviews to improve software quality continuously, see Figure 1: We perceived quality control as a simple, continuous feedback loop in which metric results and manual reviews are used to assess software quality. A quality engineer – a representative of the quality control group – provides feedback to the developers based on the differences between the current and the desired quality. However, we underestimated the amount of required manual action to create an impact. Within five years of experience as software quality consultants in different domains (insurance companies, automotive manufacturers, or engineering companies), we frequently experienced that tool

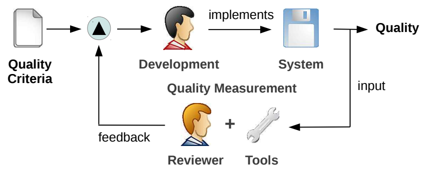

Fig. 1. The former understanding of a quality control process

support alone is not sufficient for successful quality control in practice. We have seen that most companies cannot create an impact on their code quality although they employ tools for quality measurements because the pressure to implement new features does not allow time for quality assurance: often, newly introduced tools get attention only for a short period of time, and are then forgotten. Based on our experience, quality control requires actions beyond tool support.

In this paper, we revise our view on quality control from [1] and propose an enhanced quality control process. The enhanced process combines automatic static analyses with a significantly larger amount of manual action than previously assumed to be necessary: Metrics constitute the basis but quality engineers must manually interpret metric results within their context and turn them into actionable refactoring tasks for the developers. We demonstrate the success and practicability of our process with a running case study with Munich RE which contains 32 .NET and 9 SAP systems.

II. Terms And Definitions

- A *quality criterion* comprises a metric and a threshold to evaluate the metric. A criterion can be, *e. g.*, to have a clone coverage below 10% or to have at most 30% code in long methods (*e. g.*, methods with more than 40 LoC).
- *(Quality) Findings* result from a violation of a metric threshold (*e. g.*, a long method) or from the result of a static code analysis (*e. g.*, a code clone).
- *Quality goals* describe the abstract goal of the process and provide a strategy how to deal with new and existing findings during further development: The highest goal is to have no findings at all, *i. e.*, all findings must be removed immediately. Another goal is to avoid new findings, *i. e.*, existing findings are tolerated but new findings must not be introduced. (III-B will provide more information).

III. The Enhanced Quality Control Process

Our quality control process is designed to be *transparent* (all stakeholders involved agree on the goal and consequences

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.

[1] *e. g.*, file size, method length, or nesting depth

[2] http://www.sonarqube.org/

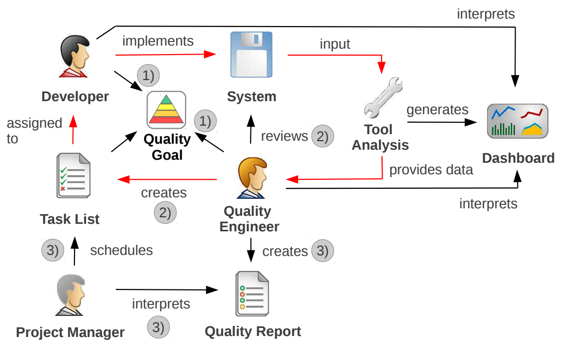

Fig. 2. The enhanced quality control process

of failures), *actionable* (measuring must cause actions) and *binding* (quality engineers, developers, and managers follow the rules they agreed on.). These following three main activities reflect these conceptual ideas and are depicted in Figure 2.

1) The quality engineer defines a quality goal and specific criteria for the underlying project in coordination with management and in agreement with the developers. Criteria are usually company-specific, the goal needs to be adapted for each system. A common definition of the quality goal and criteria makes the process transparent.
2) The quality engineer takes over responsibility to manually interpret the results of automatic analyses. He reviews the code in order to specify actionable and comprehensible refactoring tasks at implementation level for the developers which will help to reach the predefined quality goal. This makes the process actionable as metric results are turned into refactoring tasks.
3) In certain intervals, the quality engineer documents which quality criteria are fulfilled or violated in a quality report and communicates it to the management. This creates more quality awareness at management level. The project manager reviews and schedules the tasks of the quality engineer, making the process binding for management (to provide the resources) and for the developers (to implement the suggested refactorings).

In the following, we will discuss the process in detail. To demonstrate the practicability of our approach, we combine the description of the process with details from a running case study of more than four years at Munich RE. When we enhance the process description with specific information from Munich Re, we use a "@MunichRe" annotation as follows:

@Munich RE. The case study demonstrates how we apply the quality control process to 32 .NET and 9 SAP software systems mainly developed by professional software services providers on and off the premises of Munich RE. These applications comprise ∼11,000 kLoC (8,800 kLoC maintained manually, 2,200 kLoC generated) and are developed by roughly 150 to 200 developers distributed over multiple countries.

A. Quality Engineer

The quality engineer (QE) is in charge of interpreting the metric results, reviewing the system, writing the report and communicating it to developers and managers. The QE must be a skilled and experienced developer: he must have sufficient knowledge about the system and its architecture to specify useful refactoring tasks. This role can be carried out by either a member of the development team or by an external company. Based on our experience, most teams benefit from a team-external person providing neutral feedback and not being occupied by daily development. We believe that the process's success is independent from an internal or external QE, but dependent on the skills of the quality engineer.

@Munich RE. In all 41 projects of our case study, the CQSE fulfills the tasks of a quality engineer who operates on site. Due to regular interaction with the quality engineer, developers perceive him as part of the development process.

B. Quality Goals

The right quality goal for a project constitutes the backbone for a successful control process. Based on our experience, the following four goals cover the range of different project needs.

QG1 (*indifferent*) Any quality sufficient – No quality monitoring is conducted at all.

QG2 (*preserving*) No new quality findings – Existing quality findings are tolerated but new quality findings must not be created (or consolidated immediately).

QG3 (*improving*) No quality findings in modified code – Any code (method) being modified must be without findings, leading to continuous quality improvement during further development, similar as the boy scouts rule also used for extreme programming ("Always leave the campground cleaner than you found it" [7]).

QG4 (*perfect*) No quality findings at all – Quality findings must not exist in the entire project.

@Munich RE. Among the 41 projects, 2 projects are under QG1, 18 under QG2, 10 under QG3 and 11 under QG4.

C. Quality Criteria

The quality criteria depend on project type and technologies used. The quality engineer is responsible for a clear communication of the criteria to all involved parties, including third-party contractors for out-sourced software development.

@Munich RE. For .NET projects, we use 10 quality criteria based on company internal guidelines and best practices: code redundancy, structural criteria (nesting depth, method length, file size), compiler warnings, test case results, architecture violations, build stability, test coverage, and coding guidelines violations. For SAP systems, we use seven quality criteria with similar metrics but different thresholds: clone redundancy, structural criteria, architecture violations, critical warnings and guideline violations.

D. Dashboard

As part of quality control, a dashboard provides a customized view on the metric results for both developers and quality engineers and makes the quality evaluation transparent (see

Figure 2). The dashboard should support the developer to fulfill the quality goal of his projects and show him only findings relevant to the quality goal: For QG2 and QG3, we define a reference system state (baseline) and only show new findings (QG2) or new findings and findings in modified code (QG3) since then. Measuring quality relative to the baseline motivates the developers as the baseline is an accepted state of the system and only the relative system's quality is evaluated.

@Munich RE. We use ConQAT as the analysis tool and dashboard, which integrates external tools like FxCop, StyleCop, or the SAP code inspector.

E. Quality Report

In regular time intervals, the quality engineer creates a quality report for each project. The report gives an overview of the current quality status and outlines the quality trend since the last report: The report contains the interpretation of the current analysis results as well as manual code reviews. To evaluate the trend of the system's quality, the quality engineer compares the revised system to the baseline with respect to the quality goal. He discusses the results with the development team. The frequency of the quality report triggers the feedback loop and is in accordance to the release schedule. The quality engineer forwards the report to the project management to promote awareness for software quality. This guarantees that developers do not perceive quality control as an external short-term appearance, but as an internal management goal of the company.

@Munich RE. Quality reports are created for the majority of applications every three or four months. Exceptions are highly dynamic applications with five reports a year and applications with little development activities with only one report a year.

F. Task List

Based on the differences in the current findings and the quality goal, the quality engineer manually derives actionable refactoring tasks[3]. He forwards these tasks to the developers and the project manager who can decide to accept or reject a task. For accepted refactorings, the project manager defines a due date. For rejected refactorings, he discusses the reasons for rejection with the quality engineer. Before the next report is due, the quality engineer checks that developers completed the scheduled tasks appropriately. The task list constitutes the focal point in the combination of tools, quality engineers, and management. The success of the process depends on the ability of the quality engineer to create tasks that effectively increase the system's quality.

@Munich RE. The quality engineer manually inspects all findings relevant to the project's quality goal. He creates tasks for findings he considers to hamper maintainability. It remains up to him to black-list findings that are not critical for code understanding or maintenance.

[3]*e. g., Remove the redundancy between class A and B by creating a super class C and pull up methods x,y,z)*

TABLE I
SAMPLE OBJECTS FOR IMPACT ANALYSIS

Name	QG	Quality Control	Size	# Developers
A	3	4 years	200 kLoC	4
B	3	5 years	276 kLoC	5
C	3	3 years	341 kLoC	4
D	4	1.5 years	15 kLoC	2

IV. IMPACT @MUNICH RE

Proving the process' success scientifically is difficult for several reasons: First, we do not have the complete history of all systems available. Second, we cannot make any reliable prediction about the quality evolution of these systems in case our process would not have been introduced. Consequently, we provide a qualitative impact analysis on four sample systems with an evolution history of up to five years rather than a quantitative impact analysis of all 41 systems. For these samples, we are able to show the quality evolution before and after the introduction of quality control.

First, we chose the number clones, long files, long methods, and high nesting findings as one trend indicator as these quality findings require manual effort to be removed – in contrast to, *e. g.*, formatting violations that can be resolved automatically. We refer to this trend as the overall findings trend. Second, we provide the clone coverage trend and compare both trends with the system size evolution, measured in SLoC. All trends were calculated with Teamscale [6].[4]

We show these trends exemplary for the three systems with the longest available history (to be able to show a long-term impact) and with a sufficiently large development team size (to make the impact of the process independent from the behavior of a single developer). As these three systems are all QG3, we choose, in addition, one QG4 system (Table I). Figures 3–6 show the evolution of the system size in black, the findings trend in red (or gray in black-white-print), and the clone coverage in orange (or light gray in black-white-print). The quality controlled period is indicated with a vertical line for each report, *i. e.*, quality control starts with the first report date.

Figure 3 shows that our quality control has a great impact on System A: Prior to quality control, the system size grows as well as the number of findings. During quality control, the system keeps growing but the number of findings declines and the clone coverage reaches a global minimum of 5%. This shows that the quality of the system can be measurably improved even if the system keeps growing.

For System B, quality control begins in 2010. However, this system has already been in a scientific cooperation with the Technische Universitaet Muenchen since 2006, in which a dashboard for monitoring the clone coverage had been introduced. Consequently, the clone coverage decreases

[4]In contrast to ConQAT, Teamscale can incrementally analyze the history of a system including all metric trends within feasible time. Hence, although ConQAT is used as dashboard within Munich Re, we used Teamscale to calculate the metric trends.

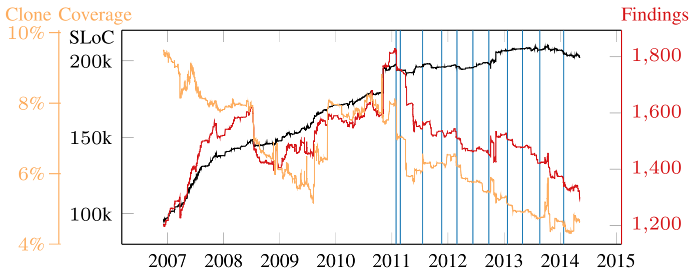

Fig. 3. System A

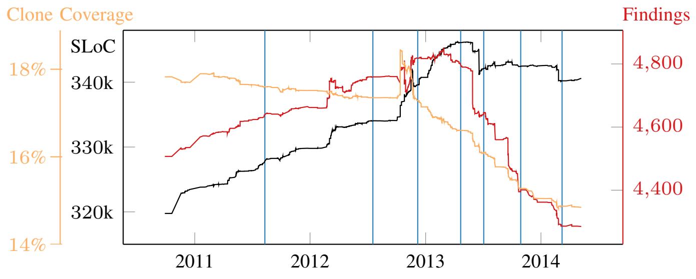

Fig. 5. System C

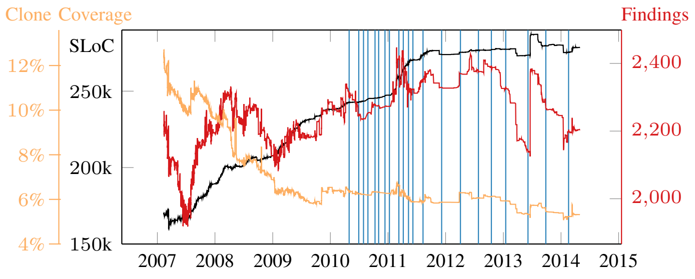

Fig. 4. System B

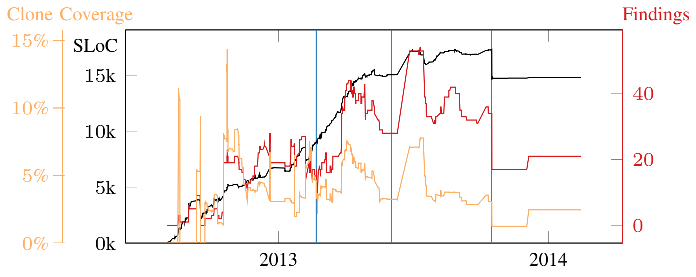

Fig. 6. System D

continuously in the available history (Figure 4). The number of findings, however, increases until mid 2012. In 2012, the project switched from QG2 to QG3. After this change, the number of findings decreases and the clone coverage settles around 6%, which is a success of the quality control. The major increase in the number of findings in 2013 is only due to an automated code refactoring introducing braces that led to threshold violations of few hundred methods. After this increase, the number of findings start decreasing again, showing the manual effort of the developers to remove findings.

For System C (Figure 5), the quality control process shows a significant impact after two years: Since the end of 2012, when the project also switched from QG2 to QG3, both the clone coverage and the overall number of findings decline. In the year before, the project transitioned between development teams and, hence, we only wrote two reports (July 2011 and July 2012).

System D (Figure 6) almost fulfills QG4 as after 1 year of development, it has only 21 findings in total and a clone coverage of 2.5%. Technically, under QG4, the system should have zero findings. However, in practice, exactly zero findings is not feasible as there are always some findings (*e. g.*, a long method to create UI objects or clones in test code) that are not a major threat to maintainability. Only a human can judge based on manual inspection of the findings whether a system still fulfills QG4, if it does not have exactly zero findings. In the case of System D, we consider 21 findings to be few and minor enough to fulfill QG4.

To summarize, our trends show that our process leads to actual measurable quality improvement. Those trends go beyond anecdotal evidence but are not sufficient to scientifically proof our method. However, Munich RE decided only recently to extend our quality control from the .NET area to all SAP development. As Munich RE develops mainly in the .NET and SAP area, most application development is now supported by quality control. The decision to extend the scope of quality control confirms that Munich Re is convinced by the benefit of quality control. Since the process has been established, maintainability issues like code cloning are now an integral part of discussions among developers and management.

V. Conclusion

Quality analyses must not be solely based on automated measurements, but need to be combined with a significant amount of human evaluation and interaction. Based on our experience, we proposed a new quality control process for which we provided a running case study of 41 industry projects. With a qualitative impact analysis at Munich RE we showed measurable, long-term quality improvements. Our process has led to measurable quality improvement and an increased maintenance awareness up to management level at Munich Re.

References

[1] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," in *IEEE Software*, 2008.
[2] D. L. Parnas, "Software aging," in *ICSE '94*.
[3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Software Eng.*, 2001.
[4] P. Johnson, "Requirement and design trade-offs in hackystat: An in-process software engineering mea- surement and analysis system," in *ESEM'07*.
[5] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessmenet," in *STEP'05*.
[6] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *ICSE'14*.
[7] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008.