# The 3C Approach for Agile Quality Assurance

André Janus
*André Janus – IT Consulting*
*Karlsruhe, Germany*
*mail@andre-janus.de*

Andreas Schmietendorf
*Berlin School of Economics and Law (HWR Berlin)*
*Berlin, Germany*
*andreas.schmietendorf@hwr-berlin.de*

Reiner Dumke
*University of Magdeburg*
*Magdeburg , Germany*
*dumke@ivs.cs.uni-magdeburg.de*

Jens Jäger
*Jens Jäger Consulting*
*Sindelfingen, Germany*
*kontakt@jensjaeger.com*

***Abstract*—Continuous Integration is an Agile Practice for the continuous integration of new Source Code into the Code Base including the automated compile, build and running of tests. From traditional Quality Assurance we know Software Metrics as a very good approach to measure Software Quality. Combining both there is a promising approach to control and ensure the internal Software Quality. This paper introduces the 3C Approach, which is an extension to the Agile Practice Continuous Integration: It adds Continuous Measurement and Continuous Improvement as subsequent Activities to CI and establishes Metric-based Quality-Gates for an Agile Quality Assurance. It was developed and proven in an Agile Maintenance and Evolution project for the Automotive Industry at T-Systems International – a large German ICT company. Within the project the approach was used for a (legacy) Java-based Web Application including the use of Open Source Tools from the Java Eco-System. But the approach is not limited to these technical boundaries as similar tools are available also for other technical platforms.**

***Keywords*-Continuous Integration; Agile; Quality Assurance; Software Metrics; Empirical Case Study**

## I. Introduction

There seems to be no traditional Quality Assurance in Agile Software Development, even though Agility promises to deliver high Quality Software. But the well-known Metrics and Measurement Approaches from traditional Quality Assurance are independent from the underlying Development Methodology. And even Agile Software Development may benefit from these approaches if combined the right way. Hence the challenge is to integrate Quality Assurance and Management into Agile Processes, which do not take Quality Assurance into account by themselves. So Quality Assurance activities have to be converted into Agile Practices or Agile Practices have to be adapted.

A famous quality-assuring technique is the Agile Practice Continuous Integration (CI). We add Continuous Measurement as subsequent activity to measure all helpful Software Metrics and we add another Activity called Continuous Improvement for the interpretation of the measures and the planning of improvement tasks for the internal Software Quality. This way we integrate traditional Quality Assurance into an Agile Software Development Process and establish Metric-based Quality-Gates for controlling the Source Code's internal Quality. The empirical results of this approach depend on a maintenance and evolution project, which may be important for the practical usage of our approach, because many Software Projects start with legacy code known as "brown-field projects" [16].

## II. Project Context

The project context is individual software development at T-Systems International - a large German ICT company. The business context is the Automotive Industry, the technical context is web application development and enterprise application integration (EAI).

### *A. Agile Software Development*

The project's Process is based on eXtreme Programming (XP) [5] [6] and was developed at T-Systems and used successfully for several years. The original XP was adjusted (and tailored) to the company´s and project's specific needs. Continuous Integration is one of XP's key Practices which is used in the project from the beginning. It means the continuous integration of new Source Code into the Code Base including the automated compile, build and running of tests. These automated steps are usually done by a Continuous Integration Engine, which assures the applications are always ready to work, i. e. ready to release. That way CI also ensures the external quality of the Software.

### *B. Maintenance and Evolution*

The project was started as a so-called "brown-field project", which means that there was already Source Code from former projects as a basis for further development. One of the project's goals was to maintain the existing Software by fixing bugs and add small improvements. The other goal was the evolution of the application, which means adding completely new features to it. A big challenge for maintaining and evolving the application was the Software Quality given by the existing Source Code. As often seen the internal Quality of the Source Code has the potential to be improved. The causes differ: It may be the lack of quality standards or quality "know-how". Sometimes it is just because the Source Code is based on a old version of the Programming language, e. g. in the project Java 1.3, which does not fit current Coding Standards.

## III. Metrics for Software Quality

The internal Software Quality is Product Quality described in ISO 25000 [14]. The Product Quality can be measured by Product Metrics, which are indicators for Quality Attributes.

### A. *Traditional Measurement Aspects and Metrics*

Traditional Metrics from LOC to the CK-Metrics [1] are well-known and understood and there is no reason why they should loose their explanatory power in Agile Software Development. There are many tools, even Open Source, to measure these Metrics. For the measurement of traditional metrics and derived metrics there are a lot of Open Source tools available. In this project we used Findbugs [7] for detecting potential programming mistakes, Checkstyle [8] for finding violations of Coding Standards and PMD [9] as a hybrid-version of the tools mentioned before.

### B. *Agile Measurement Aspects and Metrics*

With Agility came some new Metrics and Measurement Approaches [17], which are very useful in the Context of Agile Software Development. They point to the main focus of Agile Development like Testing and Continuous Integration and try to quantify the projects progress in that context. We look at the number of Tests, the Test-Coverage and number of Broken Builds. In our project the Test-Coverage was measured by the tool Cobertura [11] and the number of Tests and Broken Builds was reported by the Continuous Integration Engine CruiseControl [12].

*1) Tests:* As Agile Development enforces the Test-First Approach or Test-Driven Development (TDD), the Number of Tests is trivial but helpful Metric. Although the total number of tests has small explanatory power, it provides first insights like LOC does in the context of complexity.

*2) Test-Coverage:* The Test-Coverage measures how much of the complete Source Code is covered by Tests, i. e. how much of the Source Code is executed during Test Execution. There is a distinction between Line-Coverage and Branch-Coverage: Line-Coverage measures the code on LOC-base, Branch-Coverage measures the code based on the number of branches like if-else statements. The Test-Coverage quantifies how "good" the code is tested. It should be as high as possible. Good values from Industry projects are a Test-Coverage of 80 until 90 percent.

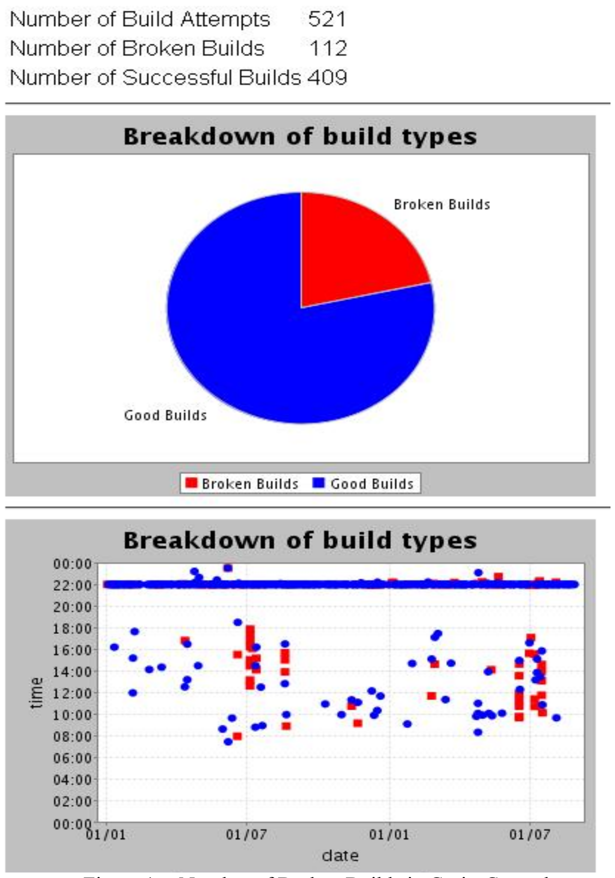
$$testCoverage = codeCoveredByTests / completeCode \quad (1)$$

$$\text{with } 0 \leq testCoverage \leq 1$$

*3) Test-Growth-Ratio:* Especially in Brown-field projects the absolute Test-Coverage is not that much expressive, when the existing Base Code had no tests. Here it makes more sense to measure the growth of the Test in relation to the growth of the Source Code. Even if there are only a few Tests, the number of Tests should be increased when the Source Code increases. But mind that the Source Code may decrease due to Refactorings. Also the number of Tests may decrease after removing functionality, which is not needed anymore.

$$testGrowthRatio = \Delta Tests / \Delta sourceCode \quad (2)$$

$$\text{with (usually) } \Delta sourceCode \geq 0 \text{ , } \Delta tests \geq 0$$



| | |
|---|---|
| Number of Build Attempts | 521 |
| Number of Broken Builds | 112 |
| Number of Successful Builds | 409 |

Figure 1. Number of Broken Builds in CruiseControl

*4) Broken Builds:* If the Continuous Integration, which is usually done by a Continuous Integration Engine, fails, this is counted as a "Broken Build" [Figure 1]. A failed Integration or broken Build shows that a Quality Requirement has not been met. Every Broken Build might be a prevented bug, that might have been released to production environment, if fixed immediately. And fixing a failed Integration has the highest priority in Agile Development. But be aware of dysfunctional effects of the Metric if it is used as a Key Performance Indicator (KPI). The Developers may intentionally cause a Broken Build. Other reasons for Broken Builds may be Infrastructure problems.

## IV. The 3C[1] Approach

Continuous Integration (CI) was naturally used just to integrate new Source Code to the Code Base, but over the years it grew a complete Eco-System of Tools and Approaches to extend CI with additional features. Many tools and Plug-In's help to continuously measure the Software's Quality with the help of Software Metrics. The main advantage of continuous measurement is to measure

1 Continuous Integration, Continuous Measurement, Continuous Improvement

the changes to Metrics and the Software Quality over time. Over time trends become visible and offer the opportunity to derive the right improvement steps, if Software Quality seems to get down. But even with a snapshot view to the measured metrics improvement steps can be derived.

## A. *Continuous Integration*

Continuous Integration is usually done by Continuous Integration Engines, which regularly check the Version Control System (VCS) - in our project Subversion/SVN [13] - for new Source Code, which a Developer may have committed after implementation in his Integrated Development Environment (IDE). If new Source Code is found, the Continuous Integration Engine compiles the Source code, builds the application and runs the tests - in case of Java applications often the JUnit Framework [10] is used for these dynamic Code Checks. If compiling, building or test execution has any errors the Continuous Integration Engine notifies the Developer(s), that have committed Source Code since the last successful integration. Now it is the highest priority for the developers to find and fix the problem.

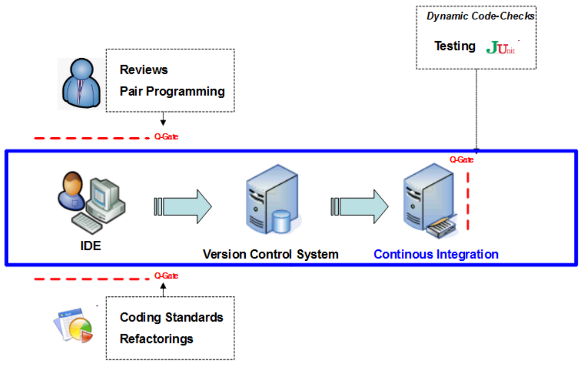


Figure 2. Continuous Integration

In [Figure 2] the CI-Infrastructure of the project at T-Systems is shown: In the center is the VCS and the CI-Engine fulfilling the steps described above and serving as a Quality-Gate by assuring that the application always can be compiled, build and pass the tests, in other words: New Source Code is always well-integrated. To avoid breaking the Build Agile Software Development provides two other Quality-Gates:

On the one hand there are the Practices Pair Programming (and Reviews), which are two Developers implementing together and controlling each other. Problems are found and fixed and structural improvements (Refactorings) are done before committing the Source Code to the VCS. On the other hand there are Formatting Rules within the IDE, which represent the Coding Standards and the Coding Style Guide. It is also possible to use IDE-versions of the metrics tools mentioned above. Developers may find violations that way before committing the Source code to the VCS. In addition to implementing new features the developer restructures the code without changing it's functionality (refactoring).

## B. *Continuous Measurement*

After the execution and successfull passing of the tests it is a common practice to get some more information out of the Source Code by adding various measurement tools in the tool chain. With these tools many static Code Checks like traditional metrics or Coding Style Guide violations can be measured. It is possible to add thresholds for certain metrics. If a threshold is exceeded the build breaks. And just like a failing test this causes the developer team to remove the issue with highest priority. This way Continuous Integration helps assuring the internal Software Quality [3] [4] by establishing another (automated) Quality Gate (Q-Gate). The direct feedback from the CI-Engine facilitates the acceptance of measurement by the developers. As they are notified immediately they get a better feeling about the outcome of their work and they also can react immediately and do necessary changes. This way the developers get the safety to perform even bigger refactorings instead of just “hacking” new feature into a design which does not meet the actual requirements and will result in technical debt on the long run.

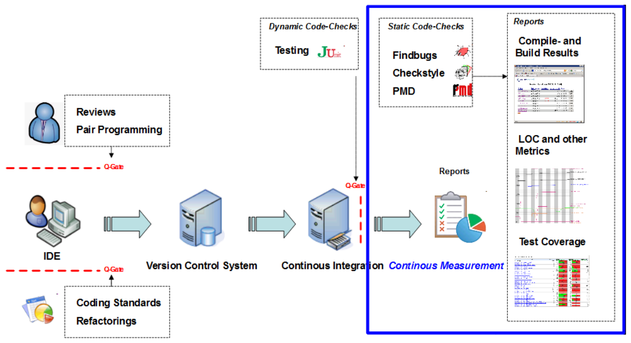


Figure 3. Continuous Measurement

In the Java Eco-System there are many Tools like Findbugs, PMD and Checkstyle – which are the ones used in the project at T-Systems - that can be integrated into an Continuous Integration like CruiseControl. The generated report built by the Continuous Measurement shown in [Figure 3] contains many information like LOC, number of packages, classes and methods. There are also reports for other artifacts like LOC in the JSPs (Java Server Pages), which represent the GUI in this web application as opposed to the LOC of the pure Java Code. During the tests the number of tests and the test coverage is measured. In our project the tool Cobertura is used for test coverage. With the number of tests and the LOC the Test-Growth-Ratio is calculated.

The measurement results are put into graphs so you can see the changes of the measures over time. For that purpose a tool (cockpit) was developed at T-Systems, which actually shows the following results:

- Total Lines (Java): LOC of the Java-based code
- Effective Lines (Java): LOC of the Java-based code without empty lines or brackets
- Total Lines (JSP): LOC of the Java Server Pages, i. e. the GUI code artifacts

- JUnit Tests: Number of the JUnit Tests
- Checkstyle error/warning/info: Violations of Checkstyle-Rules with severity error/warning/info
- Findbugs Priority 1/2/3: Violations of Findbugs-Rules with priority 1/2/3
- PMD Priority 1/2/3: Violations of PMD-Rules with priority 1/2/3

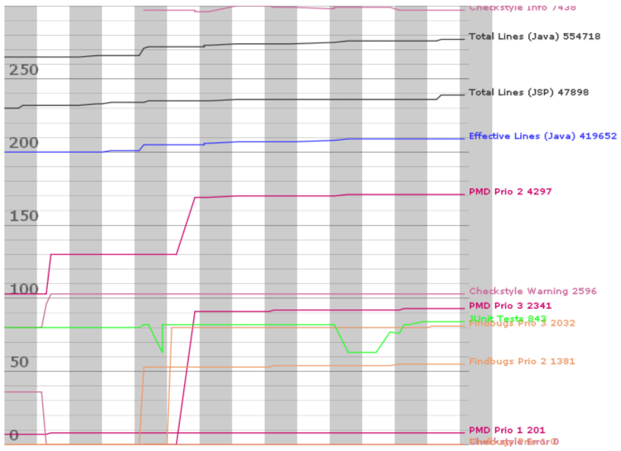


Figure 4. Cockpit results graph

In [Figure 4] it is clear to see, how the Source Code grows continuously over time as new features are added. Also the number of tests grows with the Source Code. The decrease in the middle was caused by large Code Generation activity to implement an interface to an external system. The Zero-Lines at the bottom are the Priority 1 Violations from Findbugs and the Checkstyle-Violations with severity error, which would cause the Build to break. This prevents the existence of the critical violations. All other violations are just measured and have to be interpreted the Quality Manager – which is a role, not a person and may be a distinguished member of the development team. The rapid increase of some of the violations is a result of the improvements of the measurement: Some Code Artifacts (and their violations) were just not part of the Continuous Measurement.

### C. *Continuous Improvement*

As described before the results call for manual interpretation, especially to deduct improvement steps. The results also shows that the Quality Manager should be part of the development team to interpret the results correctly. The Quality Manager plans necessary Refactorings or changes to the Coding Standards or the Coding Style Guide. This way the Quality Manager assumes the role of another Quality Gate as he can engage, if the results from the reports require it. Besides the manual analysis the Quality Manager can define new thresholds for certain metrics or add new metrics to be measured. If bigger Refactorings are necessary to meet the thresholds, the Quality Manager plans Refactorings taking into account the ongoing implementation of new features. All these Activities are called Continuous Improvement [Figure 5].

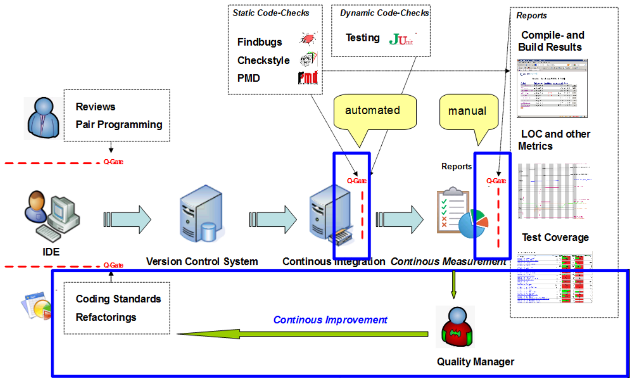


Figure 5. Continuous Improvement

To archieve concrete improvements and determine the necessary metrics the GQM (Goal-Question-Metric)-Approach is used [2] [15]. An example from the project is the removal of Findbugs-Violations with priority 1. The (Improvement)-Goal was the reduction of errors and faults in the production environment. A question to this goal is which programming bugs could be detected before releasing the software to the production environment. A metric that could possible answer that question is the number of Findbugs-Violations with priority 1. As Findbugs-Violations are only potential bugs, this metric is just an indicator, but as the priority 1 shows it is a very strong indicator. The measurement of Findbugs-Violations with priority 1 was just implemented within the CI-Engine, so the bugs could easily be identified, the effort of their removal could by estimated and planed for the next iteration. After the removal of the Findbugs-Violations with priority 1, the new threshold was added to the CI-Engine. This prevents new Findbugs Priority 1's to get into the Source Code, because every detection of a Priority 1 Bug would cause the Build to break, and the developers would immediately fix it. This way the Quality Gate Continuous Integration was extended by a new check for Findbugs Priority 1 bugs.

## V. Conclusion

Agile Software Development and especially the Agile Practice CI offers new opportunities for traditional Software Measurement - instead of making it obsolete. It provides a measurement infrastructure for the Continuous Measurement which measures both traditional and Agile metrics. And it shows the shift of metrics and quality attributes over time and leads to proper and also “Continuous” Improvement Activities selected via the GQM-Approach. The improvement process does not only lead to selective improvements but to automated Quality Gates to preserve the improvements enduringly.

Especially in the context of maintenance and evolution of legacy software there is a need for analysing and planning for improvements with care to balance efforts and costs of the improvements to their outcome. Using this 3C Approach the observed project could significantly improve its internal Software Quality and establish “in-process” Quality Gates to assure the Quality also for future development.

## References

[1] S. R. Chidkamber, C. F. Kemere, 1993, A Metric Suite for Object-Oriented Programming, MIT Sloan School of Management

[2] Rombach, H.D., Vasili, V.R., Quantitative Software-Qualitätssicherung, Informatik-Spektrum 19/1987, Pages 145-158

[3] M. Hua et al., Hows does agility ensure quality, National CT Australia Ltd. And Univeristy of New south Wales, Australia

[4] M. Hua et al., Software Quality and Agile Methods, National CT Australia Ltd. And Univeristy of New south Wales, Australia

[5] K. Beck, 1999, Extreme Programming explained: Embrace Change, 1st Edition, Addison-Wesley Professional

[6] K. Beck, C. Andres, 2004, Extreme Programming explained: Embrace Change, 2nd Edition, Addison-Wesley Professional

[7] Findbugs – Find Bug in Java Programs, http://findbugs.sourceforge.net/

[8] Checkstyle, http://checkstyle.sourceforge.net/

[9] PMD, http://pmd.sourceforge.net/

[10] JUnit, http://www.junit.org/

[11] Cobertura, http://cobertura.sourceforge.net/

[12] CruiseControl, http://cruisecontrol.sourceforge.net/

[13] Subversion (SVN), http://subversion.apache.org/

[14] ISO-Norm 25000, http://www.iso.org/

[15] Messen nach der GQM Methode, http://software-kompetenz.org/?17030

[16] Brownfield, http://en.wikipedia.org/wiki/Brownfield_%28software_development%29

[17] A. Schmietendorf, C. Wille, 2012, “Bedeutung der Softwaremessung bei agilen Projektparadigmen – eine Bestandsaufnahme” from “Beiträge zum empirischen Software Engineering – eine Bilanz” in “Magdeburger Schriften zum empirischen Software Engineering”, Shaker-Verlag Aachen