Agile Metrics for a University Software Engineering Course

Christoph Matthies, Thomas Kowark, Matthias Uflacker, and Hasso Plattner
Hasso Plattner Institute, University of Potsdam
August-Bebel-Str. 88
Potsdam, Germany
Email: {firstname.lastname}@hpi.de

***Abstract*—Teaching agile software development by pairing lectures with hands-on projects has become the norm. This approach poses the problem of grading and evaluating practical project work as well as process conformance during development. Yet, few best practices exist for measuring the success of students in implementing agile practices. Most university courses rely on observations during the course or final oral exams. In this paper, we propose a set of metrics which give insights into the adherence to agile practices in teams. The metrics identify instances in development data, e.g. commits or user stories, where agile processes were not followed. The identified violations can serve as starting points for further investigation and team discussions. With contextual knowledge of the violation, the executed process or the metric itself can be refined. The metrics reflect our experiences with running a software engineering course over the last five years. They measure aspects which students frequently have issues with and that diminish process adoption and student engagement. We present the proposed metrics, which were tested in the latest course installment, alongside tutoring, lectures, and oral exams.**

***Index Terms*—Metrics, Computer engineering, Assessment tools, Capstone projects, Higher education**

I. Introduction

In order to provide feedback to students and educators on how well Scrum and agile best practices are followed in a team, the day-to-day development process needs to be assessed. We propose objective, automated *conformance metrics* which can perform this assessment, complementing proven techniques, such as assessments by tutors or exams. Conformance metrics rely on collected development data, e.g. commits or issues, which are created during regular development activities. This ensures that established workflows do not need to be adapted and additional documentation overhead for students is avoided. Conformance metrics measure to what degree an executed process is in agreement with previously defined ones, i.e. practices recommended by agile methodologies such as Scrum or XP.

Our approach detects and quantifies instances where the executed process deviates from the defined one. These instances, i.e. patterns in the collected data that do not comply with the details of the process, are referred to as *violations*. Violations can reveal problem areas in the executed process, that need special attention. This approach is comparable to test coverage tools and the Lint [1] tool, which do not guarantee the absence of defects, but identify weaknesses.

II. Conformance Metrics

Conformance metrics follow the iterative model described in Figure 1, adapted from Zazworka et al. [2]: conformance metrics are defined, violations are detected, the context of these violations is researched and measures are taken to prevent future violations.

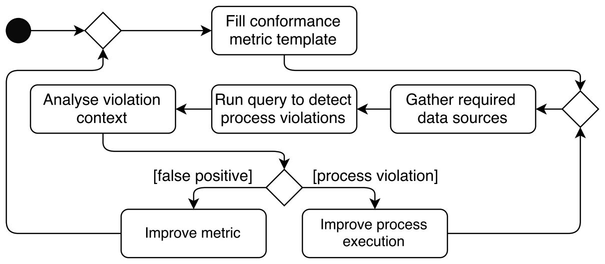
Fill conformance metric template
Analyse violation context
Run query to detect process violations
Gather required data sources
[false positive]
[process violation]
Improve metric
Improve process execution

Fig. 1. Activity diagram of the iterative lifecycle of conformance metrics.

A. Definition

In order to create a conformance metric there must be a common understanding of the practice that should be executed and measured. This involves both those who have knowledge and experience in agile development, i.e. the teaching staff as well a the team, who might have previous experience and personal preferences. Agile methodologies such as XP or Scrum advise a multitude of practices, e.g. “no functionality is added early” or “all code must have unit tests”. Sletholt et al. [3] mention 35 main ones, which can serve as a starting point to select practices that are applicable in the context of a certain project. If a process is considered relevant enough to be measured and a common understanding of its details is found, this knowledge should be recorded in the form of the process conformance template, based on previous work [2], see Table I.

a) Description: The only requirement of a metric’s description is that it is detailed enough to allow defining what patterns in the collected data constitute a violation and which do not. This means the description may, but is not required to, follow formal definitions.

b) Score Calculation: In order to allow users a quick overview, the results of a metric are summarized using a rating function. It maps the violation details returned by the query into the more abstract form of a score, bounded by a high and

Name	The unique, descriptive identifier of the metric.
Synopsis	A short description of the type of violations the metric measures, e.g. “commits without tests”.
Description	Overview of the expected process, i.e. the practice which should be followed, and its advantages, with references to literature. A description of what constitutes a violation of this process should be included.
Data sources	A list of data sources the metric requires and which the *query* is based on, e.g. code repositories or issue trackers.
Query	Steps needed to extract violations from the *data sources*. Ideally, these steps can be automated, e.g. as a database query.
Rating function	Function that maps detected violations into a numerical score, indicating the degree of mismatch between the executed process and the one detailed in the *description*.
Pitfalls	Description of what the metric does *not* measure, e.g. limitations or possible misconceptions about the results of the metric.
Categories	Topics in the domain of agile software development the metric attempts at measuring, e.g. “XP practices”
Effort	How much effort collecting violations and calculating a score requires. Either low, medium or high, e.g. using an automated process on existing data sources is “low” effort. Low effort metrics should be implemented first.
Severity	Importance in the context of the project’s agile development process. How severe violations found by this metric are. Either informational, very low, low, normal or high.

TABLE I: Conformance metric template.

low value We employed percentages from 0 to 100, where 100 indicates that no violations were detected and 0 that the described practice was not followed at all. Numerical values allow the development of a metric over time to be visualized, as well the results of individual metrics to be summed into an overall team score in an iteration, e.g. a sprint. Two main types of rating function were used:

1) *Threshold function*, a linearly decreasing function, returning 0 for inputs larger than a threshold, e.g. $g(x)$ in Figure 2.1.
2) *Cut-off parabola*, returning the perfect scores for a range of optimal input values, while results for values outside of the optimal range quickly fall, e.g. $h(x)$ in Figure 2.2.

c) Classification: The severity of a conformance metric reflects the relative importance of the metric for the process. The severity determines the weight with which the metric influences the overall score of a team. We identified three main factors, which influenced the initial severity rating:

1) **Importance in agile methodologies** Core elements of agile methodologies, e.g. working in iterations, were considered to be of high severity. Practices that are found in multiple methodologies, e.g. Scrum and XP, were rated more severe.
2) **Importance for team** Agile development practices need to be adapted to a team’s context. Even established practices can be of low importance to a particular team.
3) **Confidence in generally established values** While there is some level of consensus on agile core practices [3], there is often little consensus on what values constitute a violation. Until reasonable values for a team are found through iterative refinement, the severity of metrics would initially be low.

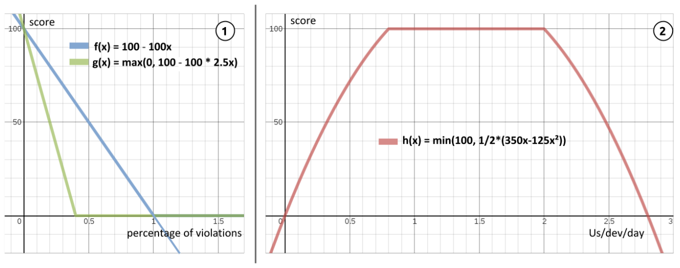

Fig. 2. Graphs of examples of the two main types of rating functions, threshold (1) and cut-off parabola (2), mapping the output of conformance metrics to a score between 0 and 100.

B. Query Execution

After development data has been collected from the defined data sources, the query part of the metrics can be used to extract process violations, see Listing 1. The rating function then outputs a list of process violations by teams and iteration as well as a score, indicating the seriousness of the extracted violations.

```
MATCH (m:Milestone)-[]-(i:Issue)-[]-(l:Label)
WHERE m.title = "Sprint 12"
and m.due_on<timestamp() AND i.current_state="open"
WITH m, m.title as Sprint, collect(DISTINCT i) AS Issues,
    count(DISTINCT i) AS Amount
MATCH m-[:milestone]-(j:Issue)
WITH Issues, count(j) AS Total, Amount, m.title as Sprint
RETURN Sprint, Amount, Issues, Total,
    Amount/Total AS Percent
```

Sprint	Amount	Issues	Total	Percent
Sprint 12	2	#129, #135	10	0.2

Listing 1: Example of a Neo4J graph database query [4], as well as an example result, of user stories that were part of sprints before Sprint 12 but are not marked as done.

C. Context Analysis

In order to gather information about the quality of violations, e.g. identifying false positives, additional information on detected violations is needed. For example, the change history of a user story can be viewed or directly discussed with the involved developers.

D. Improvement

A major part of the lifecycle of conformance metrics is their continuous improvement; this means ensuring that the metrics fit the project and team context. Especially the amount of false positives, i.e. process violations that were detected but do not actually pose a problem for a development team, need to be minimized. The amount of real violations, i.e. true positives, can be reduced by modifying the query or process description, thereby adjusting the metric to better fit the executed process, or by making sure the defined process is followed more closely, e.g. by additional training or tutoring [5].
The knowledge that was gathered in the improvement step can then be made explicit again in the first step of the lifecycle, by editing the conformance template.

III. Conformance Metrics Details

While the conformance metrics presented in this section are based on common agile practices, they are often tailored to the specific set of Scrum, agile and organizational practices that we focused on in student courses. We therefore do not include the query or the rating function, as these are not universal. They are available in a open-source licensed repository[1]. The metrics in this section are divided by their categories.

a) XP Practices: Metrics measuring violations of Extreme Programming development practices.

Name: **Collective Code Ownership**

Synopsis: Code which is edited heavily by few developers.

Severity:	*Effort*:	*Data Source*:
Normal	Low	Version control

Description: Collective Code Ownership, as defined by Beck [6] states that "every programmer improves any code anywhere in the system at any time if they see the opportunity". It is one of the core extreme programming practices. Closely related is the "bus number" [7], which is the number of developers that a project would need to lose to halt its progress. It measures the concentration of knowledge about software components in individual team members. Following the practice of Collective Code Ownership can help every developer work on any user story. This metric finds files that had many edits by only few authors. The more of these there are, the less the practice of Collective Code Ownership was followed.

Name: **Test-Later Development**

Synopsis: Increasing code complexity while decreasing code coverage.

Severity:	*Effort*:	*Data Source*:
Normal	Medium	version control, code coverage stats

Description: In TDD, an automated test is written before the code that makes it pass. This is followed by a refactoring step. Following TDD can have a positive effect on system design and assures that all code is always tested [8]. Kniberg states "You can take my house and my TV and my dog, but dont try to stop me from doing TDD!" [9]. TDD is also related to the XP practice of *YAGNI* (you ain't gonna need it) [10]. Tests act as a reminder to work on the current story. This metric identifies commits where TDD was not followed, i.e. commits which introduced additional complexity to the system, but led to decreased code coverage.

[1]https://github.com/hpi-epic/ScrumLint

Name: **Huge User Stories**

Synopsis: User stories that are unusually large.

Severity:	*Effort*:	*Data Source*:
Low	Low	User story tracker

Description: User stories should be small enough to get a quick overview of the work to be done, but should contain enough information to allow developers to estimate it. The text of a user story should fit on an index card [11]. If a user story is much longer than the average this might be an indicator that it is too large, was hard to estimate and should be split [10]. The user stories identified by this metric are significantly above the average length of stories in the sprint or have many times the amount of tasks of other stories.

b) Backlog Maintenance: Metrics attempting to measure violations concerning both product and sprint backlogs.

Name: **One Story, Multiple Backlogs**

Synopsis: User stories that were in multiple sprint backlogs.

Severity:	*Effort*:	*Data Source*:
High	Low	User story tracker

Description: Ideally, a sprint backlog contains exactly as many user stories as the team can complete in an iteration [12], so that at the end of the sprint all user stories in the sprint backlog are completed. This ensures the ability to plan the software's development and enables teams to build on the finished functionality in the next sprint. However, sometimes, at the end of the sprint not all stories conform to the "Definition of Done" [9]. These user stories are then carried over to the next sprint, if the product owner still considers them a priority. A story that spans multiple sprints can be a blocker for other teams that depend on it. This metric identifies user stories that were assigned to the sprint backlog of multiple sprints. The percentage of offending "neverending" stories should be minimal.

Name: **Duplicates**

Synopsis: User Stories which are suspected duplicates.

Severity:	*Effort*:	*Data Source*:
Very Low	Low	User story tracker

Pitfalls: This metric relies on developers or teaching staff tagging user stories as duplicates. There might be additional duplicates that were not tagged.

Description: User stories are the main tool of specifying what will be done in a sprint [11]. User stories should not overlap in described functionality, as there is the risk of features being developed twice if these user stories are given to different teams in the same sprint. This metric identifies user stories that were marked as possible duplicates by developers.

c) Developer Productivity: Metrics dealing with topics such as how work is structured, how it is assigned and the workload of developers.

Name: **At the Last Minute**

Synopsis: Commits shortly before sprint deadline.		
Severity: Normal	*Effort*: Low	*Data Source*: Version control

Description: Work in an agile project should follow a "sustainable, measurable, predictable pace" [13] and overtime should be avoided [6]. The software at the end of the sprint should be as completed, tested and integrated as possible. If work is slanted towards the end of the sprint and code is committed at the very last minute, this can cause a range of problems: Scrum meetings might be ineffective, due to lack of content, blockers for or by other teams can not be communicated in a timely fashion and code review through other developers becomes more difficult. This metric measures commits that were made during the last minutes before sprint deadline. The more of these there are, the less likely it is that a sustainable pace was followed.

Name: **No Committing**

Synopsis: Average amount of commits per developer.		
Severity: Normal	*Effort*: Low	*Data Source*: Version control

Description: The rule of *Check in Early, Check in Often* is encourages small patch sizes [14]. Jeff Atwood, co-founder of Stack Overflow, considers it a "golden rule of source control". He states that from a team member's viewpoint, "if the code isn't checked into source control, it doesn't exist" [15]. Committing finished functionality frequently is also a requirement for continuous integration and delivery called for in the principles of the agile manifesto [16]. Committing often allows coworkers to build on functionality, review the code and makes version control and merging easier. This metric measures the average amount of commits that were made by the developers of a team over the course of a sprint. Generally, the more commits were made, the better, however, they should represent working increments of the software.

Name: **Daily User Story Amount**

Synopsis: Average amount of user stories a developer is assigned per day.		
Severity: Low	*Effort*: Low	*Data Source*: User story tracker

Description: User stories should conform to the *INVEST* acronym (independent, negotiable, valuable, estimable, small, testable). Small has been defined to mean a few person-days to a few person-weeks [17]. Cohn does not state absolute values but explains that "the ultimate determination of whether a story is appropriately sized is based on the team, its capabilities, and the technologies in use" [11]. While the amount of user stories a developer should be able to finish per day is hard to state generally, working on multiple stories per day results in increased context switching overhead [18]. This metric measures the average amount of user stories a developer would have to finish every day, given constant productivity. If this number is high, it is possible that the requirements of the Definition of Done, deployment, communication and context switching overhead were underrated and the Sprint Backlog is too full.

Name: **Fast pull requests**

Synopsis: Pull requests that were closed quickly without comments.		
Severity: High	*Effort*: Low	*Data Source*: Pull Requests

Description: Pull requests can be a tool to help inform team members what functionality is added in a collection of commits. It allows team members and stakeholders to comment and perform code review. According to Boehm and Basili, code reviews by peers catch around 60% of defects [19]. Furthermore, continuous integration services can run the proposed changes, making sure all tests pass, before code is merged. If pull requests are merged in a short timespan without anyone commenting, this hints at many of these possibilities remaining unused. This metric identifies pull requests that were closed quickly and had no comments. The more of these "speedy pulls" are found, as a percentage of all pull requests, the worse the score.

IV. Conclusion

Using the presented conformance metrics, violations for all teams in all sprints could be extracted from data gathered in the 2014/15 installment of our undergraduate agile software development course with 38 participants. As the presented metrics rely solely on data, they allow a more unbiased view of teams. By providing the offending development artifacts, the root causes of violations could be established, such as bad communication between teams involved with user management. All metrics could be used to identify instances where tutor intervention would have been helpful. In some cases, severe violations were found that were missed by tutors, such as a very complex, wrongly prioritized user story, that had been in the sprint backlog of all sprints. As it was one user story among hundreds it was missed by manual analyses. As such, we see this data-driven approach as a good supplement to the usually employed assessment techniques in undergraduate student software engineering capstone projects. Future work includes employing the presented metrics in the next installments of project courses as well as continuously improving them in order to better reflect project reality and eliminate false positives.

References

[1] S. C. Johnson, "Lint, a C Program Checker," *Comp. Sci. Tech. Rep*, pp. 78–1273, 1978.

[2] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, "Are Developers Complying with the Process: An XP Study," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 14.

[3] M. T. Sletholt, J. E. Hannay, D. Pfahl, and H. P. Langtangen, "What Do We Know about Scientific Software Development's Agile Practices?" *Computing in Science and Engineering*, vol. 14, no. 2012, pp. 24–36, 2012.

[4] F. Holzschuher and R. Peinl, "Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 195–204.

[5] C. Matthies, T. Kowark, K. Richly, M. Uflacker, and H. Plattner, "How Surveys, Tutors, and Software Help to Assess Scrum Adoption in a Classroom Software Engineering Project," in *The 38th International Conference on Software Engineering (ICSE)*, Austin, TX, 2016.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[7] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *Product-Focused Software Process Improvement*. Springer, 2011, pp. 337–351.

[8] L. Madeyski, *Test-Driven Development: An Empirical Evaluation of Agile Practice*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[9] H. Kniberg, *Scrum and XP from the Trenches*. C4Media, 2007.

[10] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Addison-Wesley Professional, 2001.

[11] M. Cohn, *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004, vol. 1.

[12] K. Schwaber and J. Sutherland, "The Scrum Guide," 2013. [Online]. Available: http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf

[13] D. Wells, "The Rules of Extreme Programming," 1999. [Online]. Available: http://www.extremeprogramming.org/rules.html

[14] A. Bosu, "Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study," in *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings*. Hyderabad, India: ACM, 2014, pp. 736–738.

[15] J. Atwood, "Check In Early, Check In Often," 2008. [Online]. Available: http://blog.codinghorror.com/check-in-early-check-in-often/

[16] K. Beck, M. Beedle, A. Van Bennekum, W. Cockburn, Alistair Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, and R. Jeffries, "Agile Manifesto," 2001. [Online]. Available: http://agilemanifesto.org/

[17] B. Wake, "INVEST in Good Stories, and SMART Tasks," 2003. [Online]. Available: http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[18] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane, "Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined," in *Proceedings of the 25th international Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 641–646.

[19] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer*, vol. 34, pp. 135–137, 2001.