List of Implemented Conformance Metrics

The metrics that were implemented in ScrumLint, are presented in this section grouped by their main category. The *rating functions* of the presented metrics return scores in the range of 0 to 100, implying 100% or 0% conformance with the described process, respectively. The *weight* variables represent weighting coefficient, influencing how much violations impact the score of the metric. In a similar fashion, *threshold* variables are used to adapt the function to the project's context, fine-tuning what values constitute a violation in the context of the metric.

XP Practices Metrics in this category attempt to measure violations of XP development practices.

Name: **Personal Code Ownership**			*1*
Synopsis: Code which is edited heavily by few developers			
Category: XP Practices	*Severity*: Normal	*Effort*: Low	*Data Source*: Version control
Rating function: $max(0, (100 - \#violations * weight))$, $\#violations$ = amount of files with $threshold_e$ edits by $threshold_a$ authors.			

Description: Collective Code Ownership, as defined by Kent Beck [1] 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" [2], 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: **Introducing: Untested Complexity!**			*2*
Synopsis: Increasing code complexity while decreasing code coverage.			
Category: XP Practices	*Severity*: Normal	*Effort*: Medium	*Data Source*: version control, code coverage statistics
Rating function: $max(0, 100 - (\#violations \div \#commits * 100 * weight))$, $\#violations$ = amount commits increasing complexity & decreasing coverage.			

Description: In TDD, an automated test is written first, then 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. Kniberg states "This [TDD], to me, is more important than both Scrum and XP. You can take my house and my TV and my dog, but don't try to stop me from doing TDD!" [3]. TDD is also related to the XP practice of *YAGNI* (you ain't gonna need it) [4], tests act as a reminder to work on the current story, not on something that might be helpful in the future.

This metric identifies commits where TDD was not followed, i.e. where tests were not written before the implementation, i.e. commits which introduced additional complexity to the system, but decreased code coverage.

Name: **Scary Monster Stories**			*3*
Synopsis: User stories that are unusually large			
Category: XP Practices	*Severity*: Low	*Effort*: Low	*Data Source*: User story tracker
Rating function: $max(0, 100 - (\#violations * weight))$, $\#violations$ = amount user stories larger than $threshold_{length}$ multiplied with average length of user stories or with $threshold_{check}$ multiplied with average amount of checkboxes of user stories.			

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 [5]. 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 [4].

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 checkboxes of other stories.

Backlog Maintenance These metrics attempt to measure the violations concerning the maintenance of the backlog. Both the sprint backlogs of iterations as well as the product backlog are inspected.

Name: **The Neverending Story**			*4*
Synopsis: User stories that were in multiple sprint backlogs.			
Category: Backlog Maintenance	*Severity*: High	*Effort*: Low	*Data Source*: User story tracker
Rating function: $max(0, 100 - (\frac{\#violations}{\#totalUS} * 100 * AvgInSprints * weight))$, $\#violations$ = amount of user stories in more than $threshold_{amount}$ sprints, $\#totalUS$ = total amount of user stories in the Sprint Backlog, $AvgInSprints$ = average amount of sprint backlogs the violations were in.			

Description: Ideally, a sprint backlog contains exactly as many user stories as the team can complete in the iteration [6], meaning that at the end of the sprint all user stories in the sprint backlog are finished. 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" [3]. 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, with or without commits referencing it. The percentage of offending "neverending" stories should be minimal.

Name: **Lottie and Lisa**[1]			*5*
Synopsis: User Stories and issues which are suspected duplicates.			
Category: Backlog Maintenance	*Severity*: Very Low	*Effort*: Low	*Data Source*: User story tracker
Rating function: $max(0, 100 - ((\#duplicates \div \#totalUS) * 100 * weight))$, $\#duplicates$ = amount of user stories in the tagged as duplicates. $\#totalUS$ = total amount of user stories.			
Pitfalls: This metric relies on developers tagging user stories as duplicates. There might be additional duplicates hat were tagged as such.			
Description: User stories are the main tool of specifying what will be done in a sprint. 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.			

Developer Productivity These metrics attempt at measuring conformance to practices related to the productivity of developers. They deal with topics such as how work is structured, how it is assigned and the workload of developers.

Name: **Just-In-Time Development**			*6*
Synopsis: Commits shortly before sprint deadline			
Category: Dev. Productivity	*Severity*: Normal	*Effort*: Low	*Data Source*: Version control
Rating function: $max(0, (100 - (\frac{\#violations}{\#totalCommits} * 100 * weight))$, $\#violations$ = amount of commits made within x minutes to sprint end. $\#totalCommits$ = amount of total commits.			
Description: Work in an agile project should follow a "sustainable, measurable, predictable pace" [7] and overtime should be avoided [1]. 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.			

[1] "Das doppelte Lottchen", a German novel by Erich Kästner about twin girls.

Name: **Don't Be Afraid of Commitment** *7*

Synopsis: Average amount of commits per developer in a team.

Category:	*Severity*:	*Effort*:	*Data Source*:
Dev. Productivity	Normal	Low	Version control

Rating function: $min(100, \#commits \div \#developers * weight)$,
$\#commits$ = amount of commits,
$\#developers$ = amount of developers in a team.

Pitfalls: If the product owner is also a developer it needs to be determined how much the developer role should account for in this metric, e.g half.

Description: The rule of *Check in Early, Check in Often* is helpful to software development, encouraging small patch sizes [8]. 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" [9]. Committing finished functionality frequently is also a requirement for continuous integration and delivery called for in the principles of the agile manifesto [10]. 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 Quota** *8*

Synopsis: Average amount of user stories a developer is assigned per day.

Category:	*Severity*:	*Effort*:	*Data Source*:
Dev. Productivity	Low	Low	User story tracker

Rating function: $min(100, weight_a * quota - weight_b * quota^2)$,
$quota = \#developers \div \#sprintBacklog \div sprintLength$,
where $\#developers$ = amount of developers,
$\#sprintBacklog$ = size of the Sprint Backlog,
$sprintLength$ = length of the sprint in days.

Description: User stories should conform to the *INVEST* acronym (independent, negotiable, valuable, estimable, small, testable). Bill Wake, who originally defined the acronym, defines small to mean a few person-days to a few person-weeks [11]. Mike 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" [5]. This means the amount of user stories a developer should be able to finish per day is hard to state generally, but assigning multiple stories per day is going to result in increased context switching overhead.

This metric measures the average amount of user stories a developer would theoretically 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 at estimation and there are too many stories in the sprint.

Name: **Need for Speed(y pull requests)** *9*

Synopsis: Pull requests that were closed quickly without comments.

Category:	*Severity*:	*Effort*:	*Data Source*:
Dev. Productivity	High	Low	Pull Requests

Rating function: $max(0, 100 - (\#violations \div \#totalPRs * 100))$,
$\#violations$ = amount of pull requests that were closed quickly,
$\#totalPRs$ = total amount of 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 [12]. 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.

Teaching Scores This category contains metrics dealing with data collected by the teaching team.

Name: **Tutor Scores** *10*

Synopsis: Scores of tutors evaluating sprint meetings.

Category:	*Severity*:	*Effort*:	*Data Source*:
Teaching scores	Informational	High	Tutor rating system

Rating function: $((dedicationScore + progressScore) \div 2) * factor$,
$factor$ = the factor needed to map scores to a scale of 0-100.
$dedicationScore, progressScore$ = scores assigned to teams by tutors.

Description: Tutors sit in on the Scrum meetings of a team, especially the Sprint Planning and Review. They rate the team on two aspects: dedication and progress. The former signifies how much effort was put into implementing, customizing and trying out new Scrum practices, the latter how well the team works and has internalized Scrum values.

This metric displays the averaged tutor scores of the sprint planning and review meeting of a sprint.

Summary

This overview presented conformance metrics that were based on data of the Software Engineering II course in 2014/15 and were partly tailored to its specific characteristics. The Scrum Guide defines Scrum a framework that "functions well as a container for other techniques, methodologies, and practices" [6]. This also implies that there is no guarantee that practices and metrics that measure them and were defined in one project's context, can be directly used in a different organization's projects, which might employ different practices. As Kniberg puts it: "[...] this is only how *I* do Scrum. That doesn't mean *you* should do it exactly the same way." [3]. However, as the metrics were tailored to the course, they were also able to generate detailed and specific insights.

References

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

[2] 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.

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

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

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

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

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

[8] 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.

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

[10] 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/

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

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