Code Process Metrics in University Programming Education

Linus W. Dietz*, Robin Lichtenthäler†, Adam Tornhill‡, and Simon Harrer§
*Department of Informatics, Technical University of Munich, Germany, linus.dietz@tum.de
†Distributed Systems Group, University of Bamberg, Germany, robin.lichtenthaeler@uni-bamberg.de
‡Empear, Sweden, adam.tornhill@empear.com
§innoQ Deutschland GmbH, Germany, simon.harrer@innoq.com

***Abstract*—Code process metrics have been widely analyzed within large scale projects in the software industry. Since they reveal much about how programmers collaborate on tasks, they could also provide insights in the programming and software engineering education at universities. Thus, we investigate two courses taught at the University of Bamberg, Germany to gain insights into the success factors of student groups. However, a correlation analysis of eight metrics with the students' scores revealed only weak correlations. In a detailed analysis, we examine the trends in the data per assignment and interpret this using our knowledge of code process metrics and the courses. We conclude that the analyzed programming projects were not suitable for code process metrics to manifest themselves because of their scope and students' focus on the implementation of functionality rather than following good software engineering practices. Nevertheless, we can give practical advice on the interpretation of code process metrics of student projects and suggest analyzing projects of larger scope.**

I. Introduction

When teaching programming or practical software engineering courses, lecturers often give students advice on how to manage their group work to be successful. Such advice could be to start early so students don't miss the deadline, or to split up the tasks so everybody learns something. Intuitively, such practices deem appropriate, but do they actually lead to more successful group work? To answer this, objective metrics are needed as evidence. Code process metrics capture the development progress [1], as opposed to looking at the outcome of using static code analysis [2]. Since they have been successfully used in the software industry [3], they might be useful in programming education to give advice on how to organize the development process of student projects. As a first step towards applying code process metrics in programming education, we want to assess their explanatory power considering students' success. We mine and analyze Git repositories of two programming courses to answer our research question: *"How meaningful are code process metrics for assessing the quality of student programming assignments?"* By this, we hope to gain insights and provide recommendations to lecturers teaching such courses.

II. Method

The subject of analysis are two practical programming courses for undergraduate computer science students at the University of Bamberg: *'Advanced Java Programming' (AJP)* covering XML serialization, testing and GUIs, and *'Introduction to Parallel and Distributed Programming' (PKS)* covering systems communicating through shared memory and message passing on the Java Virtual Machine. Students typically take AJP in their third semester and PKS in their fifth.

TABLE I
Overview of the Assignments

Course	#	Technologies
AJP	1	IO and Exceptions
	2	XML mapping with JAXB and a CLI-based UI
	3	JUnit Tests and JavaDoc documentation
	4	JavaFX GUI with MVC
PKS	1	Mutexes, Semaphores, BlockingQueue
	2	Executor, ForkJoin, and Java Streams
	3	Client/server with TCP
	4	Actor model with Akka

The courses follow a similar didactic concept that has constantly been evolved since 2011 [2]. During the semester, the students submit four two-week assignments (see Table I) solved by groups of three. These assignments require the application of the previously introduced programming concepts and technologies from the lectures to solve realistic problems, such as implementing a reference manager or an issue tracker. For each assignment, the groups get a project template with a few predefined interfaces. We provide a Git repository for each group to work with and to submit their solutions. Since undergraduates in their third term are usually not proficient with version control systems, we also hold a Git tutorial at the beginning of the course, covering how to commit, push, merge, resolve conflicts, and come up with good commit messages. More advanced topics like working with feature branches or structured commit messages are not in scope of this introduction.

We grade each assignment in form of a detailed textual code review and a score between 0 and 20 points. The main part of that score accounts for functional correctness, which we check with the help of unit tests. However, we also evaluate the code quality, determined by a thorough code review. To avoid bias from one lecturer, we established a peer-review by the other lecturer. By this, the score should be an objective indicator for the quality of the solution. Over the years, we

have built a knowledge base of typical code quality issues, recently culminating into the book *Java by Comparison* [4], which we use to refer to issues in the textual code review.

A. Data Set

The data analyzed in this paper are the Git repositories of one iteration of AJP (24 groups) and PKS (14 groups) in the academic year of 2016. This results in a total of 152 submissions. All groups submitted four assignments and no group scored less than 10 points in any assignment. An assignment solution consists of all the commits related to the assignment. Each commit includes its message, the changes made, a time stamp, and the author. Each submission had at most three authors, however, this number was sometimes reduced to two, in case a student dropped out of the course. Because of their limited experience with Git, the students worked solely on the master branch. Furthermore, since the focus of the courses was not on software engineering skills, the students could freely choose how to collaborate on the assignments and we enforced no policy regarding to collaboration or the commit messages.

B. Processing and Metrics

Before mining the raw data for metrics, we performed a data cleaning step. We observed that students used different machines with varying Git configurations for their work. This resulted in multiple email identifiers for a student. Therefore, we inspected the repositories and added .mailmap[1] files to consolidate the different identifiers. Then, we did the data mining with proprietary APIs of CodeScene[2], a tool for predictive analyses and visualizations to prioritize technical debt in large-scale code bases. The tool processes the individual Git repositories together with the information about the separate assignments. We customized the analysis to calculate metrics per assignment solution and selected the following metrics:

- **Number of commits.** The total number of commits related to the specific assignment.
- **Mean author commits.** The mean number of commits per author.
- **Mean commit message length.** The mean number of characters in commit messages, excluding merge commits.
- **Number of merge commits.** The number of merges.
- **Number of bug fixes.** The number of commits with *'bugfix'* or *'fix'* in the commit message.
- **Number of refactorings.** The number of commits with *'refactor'* or *'improve'* in the commit message.
- **Author fragmentation.** A metric describing how fragmented the work on single files is across authors [5].
- **Days with commits.** The number of days with at least one commit in the assignment period.

These metrics cover the most relevant aspects of the process. Unfortunately, we could not consider the size, i.e., the number of additions and deletions of the commits, because the students imported project skeletons for each assignment and there were dependencies between the assignments. For example, in PKS the very same task had to be solved using different technologies and the students were encouraged to copy their old solution to the current assignment for comparing the performances.

[1]https://www.git-scm.com/docs/git-check-mailmap
[2]https://empear.com/

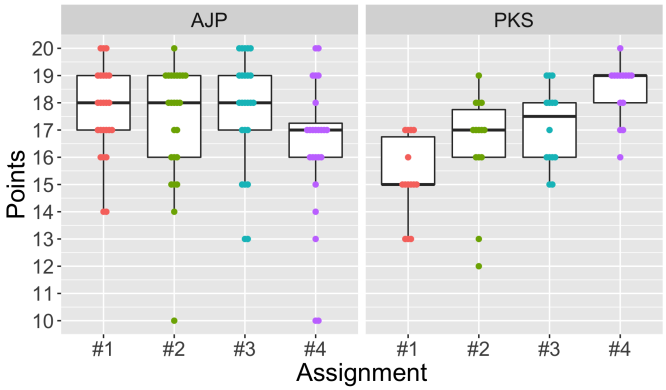

Fig. 1. Distribution of points per courses and assignments

III. Results

Before investigating the code process metrics, we display the distribution of achieved points per course and assignment in Figure 1. In AJP, the median score of 17 to 18 is quite high and homogeneous over the course, however, there is some variability with a standard deviation of 2.21. In the more advanced PKS course, the average score had a rising tendency with a median value of only 15 in the first assignment until a median of 19 in the fourth. We assume that this is due to students having little prior knowledge in concurrency programming. Additionally, the first assignment deals with low-level threading mechanisms, which require a profound understanding. The students gradually improve their performance over the course by gaining experience and because the later assignments deal with more convenient concurrency programming constructs. The standard deviation of points is 1.88.

A. Correlating Code Process Metrics with Points

To analyze the aforementioned code process metrics for correlations with the achieved points, we calculated the pairwise Pearson Correlation Coefficient (PCC) between all features over all solutions irrespective of the course. Surprisingly, we did not encounter any notable relationship between any of our metrics and points, as can be seen in the 'Overall' column of Table II.

TABLE II
Pearson Correlation Coefficient between points and features

Feature	Overall	AJP	PKS
Mean Author Fragmentation	0.01	0.09	-0.25
Mean Commit Message Length	0.20	0.35	-0.06
Mean Author Commits	0.05	0.04	-0.09
Number of Commits	0.04	0.05	-0.16
Number of Merge Commits	0.08	0.10	-0.12
Number of Bug Fixes	0.04	0.09	-0.04
Days With Commits	0.03	0.07	-0.11
Number of Refactorings	0.07	0.07	0.08

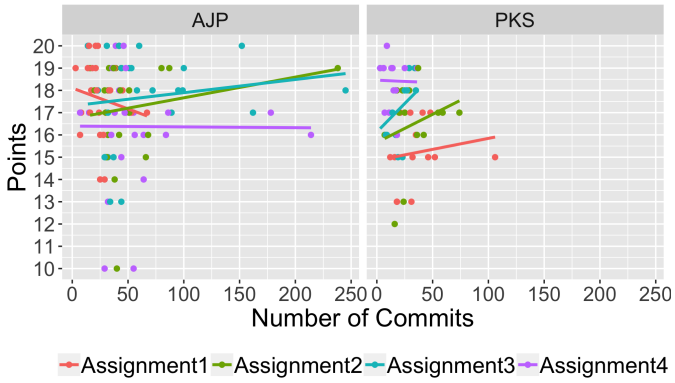

Fig. 2. The score relative to the number of commits

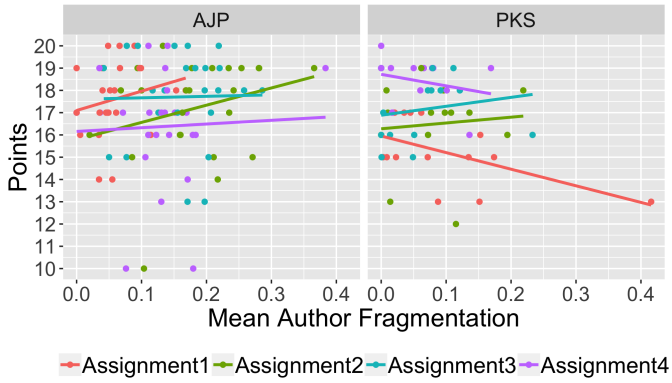

Fig. 3. The score relative to the author fragmentation

When looking at the two courses separately, however, in AJP, one can solely see a moderate positive correlation of 0.35 between the commit message length and points. Interestingly, this effect cannot be seen in PKS. There, we mainly see weak negative correlations, which is also surprising, as it seems that in contrast to AJP, more work does not lead to more points.

More effort does not necessarily mean more points. While it is generally hard to directly quantify the effort using our metrics, the combination of the number of commits and the days with commits are the best available proxy. Figure 2 shows the number of commits per course and assignment. The lines drawn on top of the data points are a linear regression model that serves as a visual aid for the detailed trends in the data. Interestingly, there is no consistent trend observable over the assignments or the course. AJP Assignment 1 has a negative trend, indicating that those groups that managed to solve the assignment with fewer commits got higher scores in that assignment. We attribute this to the prior knowledge of the students at the start of the course. In the next two assignments of AJP, the trend is positive, whereas the number of commits in the last assignment did not have an impact on the grading. In PKS, we see positive trends between both the number of commits and days with commits with the points in the first three assignments, while the last assignment shows a flat trend. Our interpretation of this matter is the following: Assignments that require much code to be written by the students benefit from more commits, while it is the other way for assignments where the framework guides the development. Recall that in Assignment 4 of AJP, the task is to write a GUI using JavaFX, and Assignment 4 of PKS is about using akka.io.

Distributing the work over a longer time span does not increase the points. Generally, we assumed that starting earlier and constantly working on the assignments, therefore, accumulating more days with a commit would increase the score. However, this is not the case. In AJP, the PCC between this feature and points was 0.07, whereas in PKS it was even a weak negative value of -0.11. We assume this has to do with the limited temporal scope of only two weeks of working on the assignments. The more experienced groups might have finished the assignment in a shorter time period and stopped working when they thought that their solution was sufficient.

Working on the same classes is not advisable. Another metric we analyzed was the author fragmentation. It measures if the Java classes were written by a sole author (zero fragmentation) or collaboratively. In AJP there was again barely a correlation of 0.09, whereas in PKS there was a weak negative PCC value of -0.25. This is somewhat in line with findings in literature, where a lower fragmentation indicates a higher quality of the software [5]. When we have a closer look at the assignments in Figure 3, however, there is again a mixed signal of PKS Assignment 1 and 4 having a negative dependency, whereas Assignment 2 and 3 are relatively stable.

Finally, we refrain from analyzing the number of bug fixes and refactorings because they were rarely used in the commits.

B. Discussion

We found no notable correlations between the analyzed code process metrics and the quality of the assignments measured via manual grading. This stands in contrast to the literature on software quality and code process metrics in industry [6]. So what makes the assignments different from real world projects?

First of all, the timeframe differs. The assignments in our courses all lasted for two weeks, whereas projects in industry span multiple months and even years. Furthermore, students focus solely on developing software, giving little thought on how to run and maintain their software. What is more, the students had a good feeling when their assignment met the functional requirements and stopped working when they were satisfied with their solution. Thus, equating the assignment scores with the notion of high-quality software is most probably not permissible in our courses.

On the other hand, it might be that code process metrics simply require a certain effort by more developers to be put into the code, which is not done by the small student groups during the short assignment period. In industry projects, maintenance work, i.e., bug fixing and refactoring, accounts for a large portion of the commits and overall quality of the software. By looking at the commit messages of the student assignments, we see that such efforts were rare. Also, communication problems become more of an issue in larger groups.

Finally, the student groups were quite new to the management of group programming tasks, especially in the third semester

course AJP. Since they could organize the development on their own, there were myriads of different strategies. We believe that this lack of organizational requirement is a key point in why we don't see clear patterns in the code process metrics.

IV. Related Work

Our approach is a contribution to learning analytics for which Greller et al. name two basic goals: prediction and reflection [7]. The commit data we analyzed has a coarse granularity compared to other work on programming education reviewed by Ihantola et al. [8], where the level of analysis is typically finer, for example key strokes. Our initial hope was that code process metrics could have some predictive power for student courses. This, however, was not the case despite several studies related to the quality and evolution of software in industry [9]. Nagappan et al. found that the structure of the development organization is a stronger predictor of defects than code metrics from static analysis [6], and Mulder et al. identified several cross-cutting concerns of doing software repository mining [10]. This paper is, thus, a parallel approach to static code analysis [2] or extensive test suites [11] for the evaluation of student assignments.

The metrics used stem from the work of Greiler et al. [12], D'Ambros et al. [1], and Tornhill [3], [13]. As an example, in industry, the author fragmentation [5] is negatively correlated with the code quality. This is supported by Greiler et al. [12], who find that the number of defects increases with the number of minor contributors in a module and Tufano et al. [14], who find that the risk of a defect increases with the number of developers who have worked on that part of the code. However, one can also go further and look at the commit metadata to capture the design degradation, as Oliva et al. did [15]. Our approach therefore combines learning analytics with insights from industry. Since in realistic projects a developer rarely programs alone, we found that the focus of our analysis should also be groups. This naturally limits us in drawing conclusions about the learning process of an individual student.

V. Conclusions

While static code analysis has often been investigated in educational settings, code process metrics from Git commits with a focus on groups represent a novel direction. We present an approach for analyzing code process metrics based on Git commits from student assignments. However, from the interpretation of our results, we cannot identify any metric that has a significant correlation with the assignment scores achieved by the students. Does this mean that code process metrics are not useful for teaching programming? From our experience, it is quite the contrary: We assume that the two courses were not a realistic setting for profiting of good coding practices. To be good software engineers in industry, students should learn how to write maintainable code, even if their code will be trashed after the semester. To establish good practices, code process metrics should play a larger role in practical software engineering courses, and could even be part of the grading. In any case, in pure programming courses with very limited timeframes code process metrics should not be used for the assessment of assignment solutions, since they are bad predictors for the score. Furthermore, when giving students guidance about how to work on programming assignments, we can give suggestions such as to start early, prefer more small commits over fewer large commits, clearly separate tasks, but they do not necessarily result in a better score.

We see time as a critical factor for the significance of code process metrics. Future work could therefore analyze development efforts with varying time frames to investigate our argument. Our paper is a first attempt at utilizing code process metrics in programming education impacted by the characteristics of the courses we considered. This means there is still potential in this topic and more research including different contexts, especially larger student projects, is desirable.

References

[1] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, *Analysing Software Repositories to Understand Software Evolution*. Berlin, Heidelberg: Springer, 2008, pp. 37–67.

[2] L. W. Dietz, J. Manner, S. Harrer, and J. Lenhard, "Teaching clean code," in *Proceedings of the 1st Workshop on Innovative Software Engineering Education*, Ulm, Germany, Mar. 2018.

[3] A. Tornhill, *Software Design X-Rays*. Pragmatic Bookshelf, 2018.

[4] S. Harrer, J. Lenhard, and L. Dietz, *Java by Comparison: Become a Java Craftsman in 70 Examples*. Pragmatic Bookshelf, Mar. 2018.

[5] M. D'Ambros, M. Lanza, and H. Gall, "Fractal figures: Visualizing development effort for cvs entities," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, Sep. 2005, pp. 1–6.

[6] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530.

[7] W. Greller and H. Drachsler, "Translating learning into numbers: A generic framework for learning analytics," *Journal of Educational Technology & Society*, vol. 15, no. 3, pp. 42–57, 2012.

[8] P. Ihantola, K. Rivers, M. Á. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, D. Toll, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, and A. Petersen, "Educational data mining and learning analytics in programming," in *Proceedings of the 2015 ITiCSE on Working Group Reports*. New York, NY, USA: ACM, 2015, pp. 41–63.

[9] M. D. Penta, "Empirical studies on software evolution: Should we (try to) claim causation?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM, 2010, pp. 2–2.

[10] F. Mulder and A. Zaidman, "Identifying cross-cutting concerns using software repository mining," in *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM, 2010, pp. 23–32.

[11] V. Pieterse, "Automated assessment of programming assignments," in *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, ser. CSERC '13. Heerlen, The Netherlands: Open Universiteit, 2013, pp. 45–56.

[12] M. Greiler, K. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, May 2015, pp. 2–12.

[13] A. Tornhill, *Your Code As a Crime Scene*. Pragmatic Bookshelf, 2016.

[14] M. Tufano, G. Bavota, D. Poshyvanyk, M. D. Penta, R. Oliveto, and A. D. Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *Journal of Software: Evolution and Process*, vol. 29, no. 1, Jun. 2016.

[15] G. A. Oliva, I. Steinmacher, I. Wiese, and M. A. Gerosa, "What can commit metadata tell us about design degradation?" in *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, ser. IWPSE 2013. New York, NY, USA: ACM, 2013, pp. 18–27.