Relying on Development Data for Software Development Processes

Christoph Matthies

Enterprise Platform and Integration Concepts
Hasso-Plattner-Institut
christoph.matthies@hpi.de

The way that developers work together, manage tasks and organize themselves impacts the software that is produced. There is thus the need to practice and uphold an effective software development process. When it comes to improving or finding weaknesses in the executed process, teams often rely on self-improvement sessions or mentoring by a knowledgeable third party. These methods rely on subjective evaluations and their results are hard to quantify. After changes are implemented it is difficult to assess their impact and whether an identified issue was fully resolved. In order to tackle these challenges we propose an approach relying on analysing data created by software development teams. Software developers create more than just code during regular development activities. Development artifacts, such as commits, tickets, wiki pages, code coverage statistics and build logs, contain information on how a team worked together. By aggregating, linking and analysing this already present data, insights into the development process of teams can be generated.

1 Overview

Specialised software analysis tools are integral parts of modern software engineering. They allow insights into the created products where manual analyses would be complex and costly. Examples for such programs include linters as well as code coverage or formal verification analyses. Most of these common tools rely on static program analysis, scrutinizing the produced source code and giving recommendations for code improvements. Data on the specifics of how the code was created is not needed and is not evaluated. This is of benefit, as no additional overhead for collecting development data is introduced, e.g. by surveying developers or making them document their efforts.

We argue that existing development data can be used to gain insights into the employed software development process. Adding administrative work is avoided, as only artifacts created during regular development activity are considered. This is enabled by the way that software engineers work: In contrast to many other engineering disciplines, software developers "self-document", i.e. they frequently produce artifacts that allow following their progress. For example, by committing code into a common software repository and updating the corresponding issue, a software engineer not only shares exactly what was done, but a variety of other information:

- **Commit metadata**, such as the commit author's email address as well as the date and time that the changes were made.

- **Commit message**, comprising the goal of the change, as well as the sentiment of the developer while she was writing it. Furthermore, the commit message can include structured information, such as an issue number (e.g. "Ref #123").
- **Issue information**, including the motivation for the change, possibly in a semi-structured "as a <user> I want to <goal> so that <benefit>" user story pattern.
- **Issue metadata**, including information on which developers were assigned or reassigned to this ticket over time, when the ticket was created and by whom, which priority it has, as well as other labels attached to it.
- **Issue comments**, made by other developers, clarifying uncertainties, requesting changes or discussing the merits of the issue's goal or associated commits.

In most development teams, the created development data is shared openly. This is especially true for open-source development. In most cases even data on the problems that occurred is readily available, through discussions in issues, wiki pages or logs of broken builds. With this wealth of available information, we can build tools that can help developers gain an overview of the *implicit* data they and their team produce and allow them to identify possible problems in the executed process.

2 Approach

In order to develop effective data-driven tools for supporting development processes access to software development teams is vital. Only with observations on how teams interact, what existing development tools are used and what data is produced, can tools be built that can attempt to be relevant.

2.1 University Courses for Data Mining

Software engineering II is a final year undergraduate software engineering course in which multiple student teams, employing the Scrum methodology, develop a single software system together. It provides a recurring opportunity to gather realistic data and develop appropriate metrics for real development teams facing real-world challenges. Especially analysis on frequent "beginner mistakes" are pertinent. Teams with little experience applying agile methodologies are likely to make mistakes, which can be quickly rectified by providing feedback or pointing to alternative practices. The course blends a hands-on development project, coaching by tutors as well as minimal lectures on agile methodologies. Figure 1 depicts the Scrum process that is being followed, including adaptations that account for the fact that students cannot devote their entire work time to the project.

In order to be as close to real-world scenarios as possible, students are consciously allowed to make mistakes and learn from them. One of the areas that is focused on are the values of the Agile Manifesto, such as "individuals and interactions over processes and tools". Students are given authority and responsibility for decisions

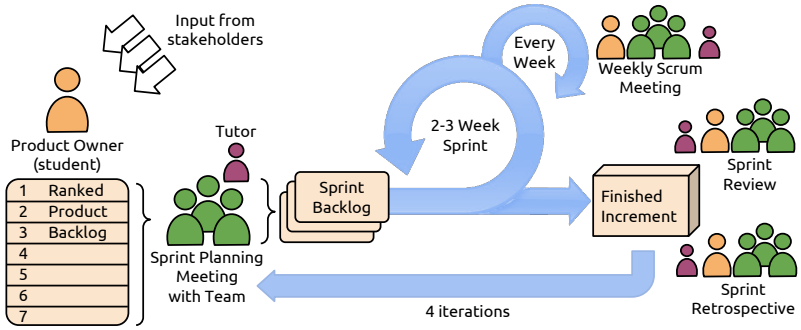

Figure 1: The modified Scrum process followed in *Software Engineering II*.

and role assignment and self-organize in their teams. These *self-organizing teams* are one of the base concepts of agile methodologies [3] and have been identified as a critical success factor for projects[1].

In order to ascertain what impact the focus on self-organizing teams that create real development data has on the satisfaction with Scrum as well as the overall course design, surveys were devised and analysed [8]. As the same survey as in a previous study by Mahnic [7] was used, the results could be compared to those of Mahnic, who employed a much stricter, more controlled course setup, featuring no explicit self-organization by students. Figure 1 lists the topics and questions of the main survey.

Both courses received consistently positive scores over all sprints for questions relating to satisfaction with the performed work as well as the Scrum methodology. These findings are in line with more extensive studies [10] on the subject.

However, the survey question directly relating to students' satisfaction and experience with letting teams self-organize only received slightly above neutral average results. Answers featured standard deviations of more than one point, a disparity that was also evident in feedback by students. While some students looked fondly on attending only a few lectures and focussing on self-organization, others would have preferred more guided tutorials and stricter organization. Reconciling these two sides in a software engineering course is an ongoing challenge. Even so, the focus of the course on self-organizing teams did not hinder the perceived learning results of students.

2.2 Data Collection

Once it is clear, which teams should be analysed and what their context is (e.g. what tools they are using), as a first step, the development artifacts that they produce have to be collected.

Modern development teams are supported by a variety of tools. These include programs which have become more or less standard, such as issue trackers and version control systems, but also a variety of other solutions, e.g. continuous integration or static code analysis.

Table 1: Main sprint survey adapted from Mahnic [7], which was filled out by students at the end of every of the four sprints of the project. Questions could be answered on a 5-point scale of "strong no" to "strong yes".

#	Question
1	**Clarity of requirements in the Product Backlog** Were the user stories in the backlog clear enough? Did the descriptions suffice to understand what the Product Owner really wanted?
2	**Effort estimation** Were the estimates of required work (story points/man-hours) of user stories adequate/realistic?
3	**Maintenance of the Sprint Backlog** Was it clear how to handle user stories? How to log performed work?
4	**Administrative workload** Does the administrative work called for by Scrum (meetings, planning, reviews, maintenance of backlog, etc.) represent a significant additional workload?
5	**Cooperation with the Scrum Master** Was the cooperation with the Scrum Master adequate/satisfactory? Did the Scrum Master contribute to the team's success?
6	**Cooperation with the Product Owner** Was the cooperation with the Product Owner adequate/satisfactory?
7	**Cooperation with other Team members** Was the cooperation with other Team members satisfactory/adequate? Did the Scrum process encourage cooperation?
8	**Workload** Was the amount of work required for the project adequate?
9	**Satisfaction with work** Are you satisfied with the work/the results of this sprint?
10	**Satisfaction with Scrum** Is the methodology useful? Would you recommend Scrum to other software developers?

In order to perform analyses on the data within these systems an *ETL process* is necessary. Data needs to be extracted, transformed into a unified data scheme and loaded into a suitable database.

A major challenge in implementing this process is keeping up to date with the development of new systems and the evolution of existing tools. Most current systems that exist for collecting development data, such as SonarQube [2] rely on connection plugins for each data source, which need to be updated when the source changes. Furthermore, they define a rigid data scheme that custom data (e.g. extra fields on a ticket) do not necessarily fit into. Lastly, it is of benefit to allow a selection of target databases that the mined and connected data can be written to, depending on the analysis use case. For example, if the social graph of developers is to be analyzed, a graph database which includes graph analysis algorithms is best. However, when text analysis is required, a document store might be superior.

DataRover [6] tackles these issues by reducing the implementation effort for ETL workflows. The querying of the data source APIs and data storage are separated. Figure 2 describes the architecture of the system. *Explorers* are each responsible for querying a single data source and represent a minimalistic connector implementation.

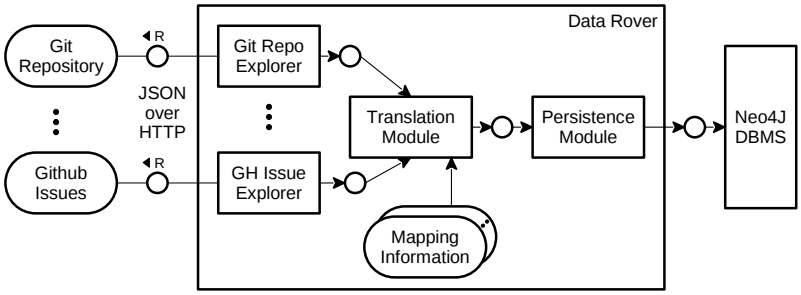

Figure 2: FMC block diagram of the architecture of the data rover.

The transformation step into a property graph, which has no previously defined data scheme, is guided by user-defined mappings. These are created through a graphical front end based on sample API JSON responses. For example, every commit could be linked to its corresponding continuous integration build by its SHA hash. A Neo4J graph database serves as an intermediate data sink that stores the connected graph of development data. The graph can then be exported to a variety of different databases to allow further analysis and the use of other tools.

Using DataRover, it is possible to create minimalistic data sets tailored to specific use cases, by using a graphical front end and only having to write querying code in case of API changes. The system's linking mechanisms further allow adding additional data sources when needed and extending existing analyses.

2.3 Development Data Analysis

Once the data of all the tools that development teams use is collected, linked and stored, it can be analysed. Our main use case, is to gain insights into the development process of student teams, to find out how well Scrum and agile best practices are followed in a team and allow targeted counteractive measures. To this end, we propose a set of nine objective, automated conformance metrics which can perform this assessment [9], complementing proven, battle-tested techniques, such as assessments by tutors or exams. We have split the developed metrics into three categories:

- **XP Practices**: Metrics measuring violations of Extreme Programming development practices.
- **Backlog Maintenance**: Metrics measuring violations concerning entries in both the product and sprint backlogs.

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

Conformance metrics attempt to extract patterns in the collected data that do not comply with the defined practices. In practice, these are processes recommended by agile methodologies such as Scrum or XP. Furthermore, we included metrics which students frequently had issues with in past instalments of the Software Engineering II course, that diminish process adoption and student engagement, and are supported in the literature. For example, a user story that is overly long and does not fit on an index card and should be split, can be identified. These violations can reveal problem areas in the executed process, that need special attention. Metrics follow the iterative model described in Figure 3, adapted from Zazworka et al. [12]. First, conformance metrics are defined using a template, containing the minimum of information that is needed in order to execute the metric and interpret results. The template is given in Figure 2. In a second step, metrics are executed and violations are detected, down to the artifact level, i.e. the actual offending artifact is extracted. Third, the context of the detected violations is researched, i.e. the how, what and why questions surrounding the artifact are discussed with the team or relevant stakeholders.

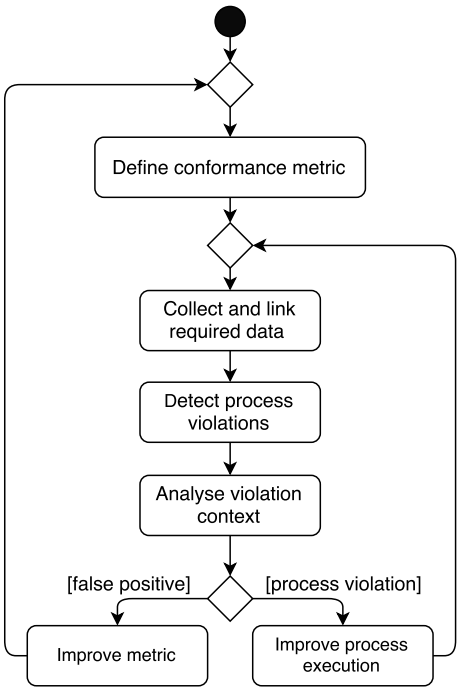

Figure 3: Activity diagram of the iterative lifecycle of conformance metrics.

Lastly, measures are taken to prevent future violations. In the case of false positives, i.e. artifacts that were detected as violations but are not considered problematic, the metric needs to be adapted. In the case of a correctly identified process violation that is considered severe enough to tackle, steps to ensure that the defined process is followed more closely in the future, e.g. by additional training or additional software tools, can be taken.

Using the proposed conformance metrics, violations for all teams in all sprints could be detected in the development data of the 2014/15 instalment of Software engineering II with 38 participants. As the specific artifact that caused a violations can be extracted by the metrics, additional research can take place to uncover the root cause of an issue. These root causes can then be tackled with additional tutor intervention. In some cases, severe violations were found that were missed by tutors, who took part in meetings. For example, a very complex, wrongly prioritized user story, that had been in the sprint backlog of all sprints. Being one user story among hundreds in the issue tracker that was being used it was missed by manual analyses.

Table 2: Conformance metric template. This template is filled for every metric that is to be measured. It contains all the information needed for an executable metric with interpretable results.

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.

As such, we see this data-driven approach using the developed metrics as an effective addition to the usually employed assessment techniques of agile teams.

3 Related Work

Due to the effort involved with analysing development data manually, multiple automatic tools, targeting specific areas, have been proposed. Examples include work by Johnson et al.[4], who proposed *Zorro*, a tool for finding violations of Test-Driven-Development practices or as well as tools for analysing development artifacts in respect to the PSP (Personal Software Process) [5]. However, with the rise of Software-as-a-Service solutions and a focus on web technologies, recent analysis tools, even in research, have moved away from native desktop applications. A good example

of this, as well as a good example of how process metrics are gaining traction is *Gitlab*,[1] a collaborative git repository hosting service on the web. Over time, Gitlab has included a full range of services, from issue tracking, code reviews, to continuous integration and continuous delivery. This gives them access to a wealth of development data for each process, along the deployment pipeline of a project. They used this data to develop process metrics called *Cycle Analytics*.[2] These measure the time it takes for an idea to pass through the different stages of a project, i.e. issue, plan, code, test, review, review and staging, starting with an issue, until code that implements the idea is deployed. These analysis are possible, as Gitlab has access to a variety of development artifacts. Without this diversity and the knowledge to link them, e.g. knowing that "Fixes#123" in a commit message relates to issue number 123, no times relevant for projects could be measured. A different approach is taken by *Commit Guru* by Rosen et al. [11]. The tool relies solely on version control information, taking a git repository on GitHub as input. Every commit is assigned a rating, placing it either in the "risky" or "not risky" categories. A risky commit is likely to contain bugs and should be reviewed. Whether a commit is determined to be risky or not is based on thirteen metrics that are calculated for each commit. These include very simple ones, such as the amount of deleted lines (higher is considered riskier) or the amount of commits the developer has recently made (less is riskier). However, some more complex metrics, such as subsystem developer experience, measuring the number of commits the developer made in the past to the subsystems touched by current commit (higher is less risky), show that meaningful statistics can already be generated from a single data source.

4 Future Work

Future work in the area of measuring process conformance by analysing development artifacts includes iterating and refining the employed metrics. As every development team is different and software development methodologies are meant to be adapted to a team's context, finding the differences in needed metrics is an interesting field of study. The related work can give very good starting points for own measurements, as these represent metrics that other teams deem relevant to their processes. Furthermore, future work will focus on how developers can be notified of possible violations. Providing a read-only web interface that needs to be regularly visited for updates, might not be the ideal solution. Providing notifications in a prompt fashion could allow new ways to interact with analysis tools, leading to greater engagement with developers.

[1] https://gitlab.com/ (last accessed 2016-10-20).
[2] https://about.gitlab.com/solutions/cycle-analytics/ (last accessed 2016-10-20).

5 Publications

Three papers were accepted in conferences in 2016:

- C. Matthies, T. Kowark, and M. Uflacker. "Teaching Agile the Agile Way – Employing Self-Organizing Teams in a University Software Engineering Course". In: *American Society for Engineering Education International Forum*. New Orleans, Louisiana: ASEE, 2016 [8].
- T. Kowark, C. Matthies, M. Uflacker, and H. Plattner. "Lightweight Collection and Storage of Software Repository Data with DataRover". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. New York, NY, USA: ACM, 2016, pages 810–815. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970286 [6].
- C. Matthies, T. Kowark, M. Uflacker, and H. Plattner. "Agile Metrics for a University Software Engineering Course". In: *46th Annual Frontiers in Education Conference*. Erie, PA: FIE, 2016 [9].

References

[1] S. Augustine. *Managing agile projects*. Prentice Hall PTR, 2005.

[2] G. Campbell and P. P. Papapetrou. *SonarQube in Action*. Manning Publications Co., 2013.

[3] T. Chow and D.-B. Cao. "A survey study of critical success factors in agile software projects". In: *Journal of Systems and Software* 81.6 (2008), pages 961–971.

[4] P. M. Johnson and H. Kou. "Automated Recognition of Test-Driven Development with Zorro." In: *AGILE*. Volume 7. Citeseer. 2007, pages 15–25.

[5] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. "Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH". In: *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*. Aug. 2004, pages 136–144. DOI: 10.1109/ISESE.2004.1334901.

[6] T. Kowark, C. Matthies, M. Uflacker, and H. Plattner. "Lightweight Collection and Storage of Software Repository Data with DataRover". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. New York, NY, USA: ACM, 2016, pages 810–815. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970286.

[7] V. Mahnic. "Teaching Scrum through Team-Project Work: Students' Perceptions and Teacher's Observations". In: *International Journal of Engineering Education* 26 (2010), pages 96–110.

[8] C. Matthies, T. Kowark, and M. Uflacker. "Teaching Agile the Agile Way – Employing Self-Organizing Teams in a University Software Engineering Course". In: *American Society for Engineering Education International Forum*. New Orleans, Louisiana: ASEE, 2016.

[9] C. Matthies, T. Kowark, M. Uflacker, and H. Plattner. "Agile Metrics for a University Software Engineering Course". In: *46th Annual Frontiers in Education Conference*. Erie, PA: FIE, 2016.

[10] G. Melnik and F. Maurer. "A cross-program investigation of students' perceptions of agile methods". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* May 2005, pages 481–488. DOI: 10.1109/ICSE.2005.1553593.

[11] C. Rosen, B. Grawi, and E. Shihab. "Commit Guru: Analytics and Risk Prediction of Software Commits". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pages 966–969. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2803183.

[12] 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, page 14.