A Field Experiment on Gamification of Code Quality in Agile Development

Christian R. Prause[1], Jan Nonnen[2], and Mark Vinkovits[1]

[1] Fraunhofer FIT, Germany
christian.prause@fit.fraunhofer.de
[2] University of Bonn, Germany
nonnen@cs.uni-bonn.de

Abstract Internal quality of software reduces development costs in the long run but is often neglected by developers. CollabReview, a web-based reputation system for improving the quality of collaboratively written source code, was introduced into an agile development team. The goal was to improve the quality of developed source code as evidenced by the amount of code entities furnished with Javadoc comments. A money prize as an extrinsic reward and peer-pressure in form of a published ranking table were tied to reputation scores. We report on the conduction of a field experiment, our observations and experiences, and relate the results to answers from concluding interviews. Although the gamification had less effect than we had hoped, our experiment teaches valuable lessons about social effects and informs the future design of similar systems.

1 Introduction

We investigate the efficacy of a reputation system as a tool to improve source code quality. According to the Oxford Dictionary, reputation is what is generally said or believed about the abilities or qualities of somebody or something. A reputation system is a software that determines a user's reputation from his actions. Scores are computed to predict future user behavior or to create peer-pressure. Reputation systems are a core component in web-based communities, where they promote well-behaving and trust (Jøsang et al., 2007).

Writing documentation is a form of well-behaving in software projects. A problem is, however, that "developers don't like to do documentation, because it has no value for them." (Selic, 2009). Source code, in particular, combines executable instructions relevant to machines with human-readable documentation. It is an important a medium of communication between humans (Dubochet, 2009). Comments, for example, include "background and decision information that cannot be derived from the code" and are "one of the most overlooked ways of improving software quality and speeding implementation" (Raskin, 2005). The developers' dislike for documenting leads to a lack of internal quality, which has become a pervasive problem in software projects (Prause, 2011). Software quality is a complex concept without a simple definition or common way of measuring it. ISO 9126-1 defines quality as Functionality, Reliability, Usability, Efficiency, Portability and Maintainability. We focus on understandability and documentation of source code as means of improving Maintainability and internal quality.

CollabReview is a reputation system for collaboratively written texts like source code. While responsibility is essential for preventing careless development and achieving quality, it is difficult to assign. CollabReview acquires responsibility information from the documents' evolution history. By making personalized data about contribution quality available, it enables self-monitoring and learning processes within a development team (Prause and Apelt, 2008). Experts believe that reputation systems can improve source code and documentation quality in agile projects but are difficult to get right (Prause and Durdik, 2012). Prause and Eisenhauer (2012) compared CollabReview's scores to actual social reputation, and found correlations of $r = 0.64$ (source code) and $r = 0.88$ (wiki). Dencheva et al. (2011) report suitability for wikis.

This paper presents a field experiment of CollabReview in a software project with the purpose of improving in-line documentation (Section 2). As Section 3 shows, the effects of the

CollabReview intervention did not have the expected effects, and with regard to this, the experiment failed. However, we interviewed our developers afterward to investigate what happened and to learn success factors for future reputation systems for quality improvement (Section 4). Section 5 discusses threats to validity. Related work is reviewed in Section 6. Section 7 summarizes the insights we gained from the "autopsy". They are a major contribution and assist the future design of similar systems. Section 8 looks out to future work and concludes the paper.

2 Experimental setup and methodology

This section describes the research methodology of our field experiment and its setup. Harrison and List (2004) define a field experiment as an experiment taking place under "natural" conditions regarding several dimensions like the subject pool or the environment the subjects operate in. They note that there is no sharp line between lab and field experiments and that there are often varying degrees of naturalness. While limiting experimental control, field experiments have the big advantage of a natural context which is essential for studying human behavior. Therefore the size of the study is small: it is the typical size of an agile team. Note that any presented students' names are pseudonyms taken from the Simpsons cartoon series. The postmortem autopsy of what went wrong based on developer feedback follows in Section 4.

2.1 XP-lab 2011

The eXtreme Programming Lab and Seminar (XP-lab) was a post-graduate teaching activity. The lab takes students into the daily work of software development, familiarizes them with agile development methodology, and generates software artifacts that are intended to be actually used in the department's work. Its realism made it a favorable environment for our field experiment. The topic was to improve a static analysis toolkit for Java software. The toolkit's analysis components are written in Prolog, while the library, its Application Programming Interface (API) and IDE integration are written in Java. Our study considered only the Java parts.

The development team consisted of instructors and ten student developers doing agile development with pair programming. The students had previously received training on the topics software engineering in general and agile methodologies in special lectures and seminars. Solid programming experience was expected. The project ran for full four weeks. Development took place in a large office space from 9am until 5pm to 6pm every day. 725 revisions were committed to the Subversion repository, including a few contributions from non-lab participants.

2.2 A pragmatic definition of internal quality

Maintainability means "the capability of the software to be modified" (ISO 9126-1, 2001). The idea of coding rules is to ease the understanding of source code by reducing distracting style noise so that it gets easier to read (King et al., 1997; Seibel, 2009; Spinellis, 2011). We acknowledge that neither maintainability nor understandability are all about rules-compliant code. For instance, identifier naming is an important part of making code understandable. However, since only style (but no understandability) checkers are readily available, following coding rules is a pragmatic decision.

How to measure internal quality was a matter of discussion prior to the lab. Consensus was that quality was to be defined through the understandability of source code. Yet the full set of Java code conventions that the Checkstyle[1] toolkit can check were not considered sensible by all discussants. Some rules were considered as too much to not hinder the effective realization of functionality, too constraining or, like "do not use tab characters for indentation", even counter-productive. Also, the organizer's wanted to discuss some rules with the developers

[1] `http://checkstyle.sourceforge.net`

while development was on-going. In the end, only rules regarding the use of Javadoc (API documenting comments) were left. As smallest common denominator, internal quality would be defined through the correct use of Javadoc comments.

Checkstyle was configured to only check for completeness of Javadoc documentation: every source code entity missing a Javadoc comment or according tags (like @param, @return or @throws) was a rule violation. The higher the density of violations (i.e. violations per line of code) in a file, the lower its quality rating. A file not missing any comments had a quality rating of +10, while higher violation densities caused lower ratings. With this definition, a file's quality rating could be arbitrarily low, so we limited it to −10.

2.3 Control and experimental phases

The arbitrary re-combination of developers in programming pairs and their small number made it infeasible to split them into a control and an experimental group. So instead, the project was split into two phases: a control and an experimental phase. First, comparison data would be collected during the earlier phase without the developers' knowing. In the second phase, after about half of the project duration, the intervention would start. The experimental phase lasted for 8 working days and served to measure the effect of CollabReview. We call the day when the intervention started "day 0".

2.4 The CollabReview intervention

CollabReview accesses the project's revision repository to determine personal reputation scores. A developer who has contributed to good files has a higher score than a developer who contributed more to rather bad files. Developers are held statistically responsible for the quality of their source code. Their score is the average quality of files that they contributed to. In the best case, if a developer has only contributed to files with a quality rating of +10, then his own score will be +10, too. The average quality is twice weighted: once by the contribution ratio of the developer for that file (more contribution means more weight) and once by size of the file (larger files have a bigger influence). For example: 75% of the lines of a file (size 1) were written by Alice, while 25% were contributed by Bob. Its quality rating is $q_1 = 8$ (good), leaving Alice and Bob with quality scores

$$s_A = \frac{0.75 \times 1 \times 8}{0.75 \times 1} = s_B = \frac{0.25 \times 1 \times 8}{0.25 \times 1} = 8$$

But Alice also solely wrote a file twice as big and with a quality rating of only poor $q_2 = 0$:

$$s_A = \frac{0.75 \times 1 \times 8 + 1 \times 2 \times 0}{0.75 \times 1 + 1 \times 2} = \frac{6 + 0}{2.75} \approx 2.2$$

If Bob also contributed 50% to a flawless file $q_3 = 10$ with size 1, he would have $s_B = 9.3$ points.

The beginning of the experimental phase was signified by an introductory email sent to all participants. It explained that developers were taking part in the experiment, how their personal scores depended on Javadoc in their code, and what Javadoc would be expected. It was noted that the measurements were not totally accurate but reflected trends.

CollabReview then sent all developers a daily email in the late afternoon. This email contained a ranking list of all developers (including their individual points), and repeated a short explanation of how developers could improve their quality ratings by writing Javadoc. In addition to the published ranking list, a 30EUR Amazon voucher was announced as a prize for the developer who would have the highest reputation score in the end. A short information on the top scores was given in the standup meeting at the end of each day. Standup meetings are used in agile development to discuss current issues and to coordinate work between team members.

There were two partly conflicting goals set for the project: the primary goal was to deliver a functional software with a hard deadline, while the secondary goal (supported through CollabReview) was to obtain high quality source code. These two goals are conflicting to some degree in the short term. In particular, we find that a developer's contribution quality and quantity were correlated at intervention start ($r = -0.70$) and end ($r = -0.43$), respectively. But although the lab was graded, neither source code quality as determined by CollabReview nor the quantity of contributions would affect a student's lab grade. Consequently, even if improving quality would cost developers their time, this would not affect their grade in neither a good nor a bad way. Developers were free to invest (almost) any amount of time into writing Javadoc comments and they were expected to document public API entities well. The only trade off between implementing and documenting was that of having less time for other implementation activities when documenting, just as in any real-world software project.

2.5 Observations during and notes on the conduction

During the daily standup meetings shortly after the CollabReview intervention had begun, the students expressed some confusion about how their reputation scores were computed. Also, in the working time further murmur on the scores was observed by the teaching staff. Still not many explicit questions were asked by the students about the scores.

A bug in the rank computation algorithm led to wrong reputation scores being published via email. Some users had actually a different rank than the one published in the daily email. Luckily, however, this mishap did not affect the winner of the prize, who was Millhouse in both cases. In fact, the mishap later turned out to be fortunate when interpreting the results.

3 Effects of the intervention

This section analyzes the reputation data that was collected during the field experiment.

3.1 Contributed code quantity

Figure 1 shows how the amount of code contributed by individual developers evolved over time; measured in 100 non-empty lines of source code (HSLOC). The figure starts at day -11, which was the first day of the project. More developers appear during the early days of the project when making their first commit to the code base. In general, the figure does not contain any surprising results: As more and more code is developed, the developers' contributions increase.

The figure does not count days on weekends. Therefore, day 0, which marks the start of the intervention, is after a little more than two weeks into the project. The project ended at day 7, one and a half weeks after the intervention start. There is a sudden increase to a high peak and then a rapid decrease to a normal level for Sanjay. This phenomenon results from copying a huge amount of code into the code base, which is later removed or edited by other developers.

3.2 Quality reputation scores

Figure 2 is more important with regard to our study. It shows how the developers' individual reputation scores for quality evolved over time. Especially in the beginning, these scores jumped up and down. The reason is that initially the developers have contributed only few code, while reputation scores are computed from the average quality. When the amount of code authored by a developer increases, each line will have less influence on the average. The smooth line shows the average score of all developers with scores weighted by their developers' quantity, i.e. a developer's score has a higher weight when that developer has contributed more code. The average score is therefore equivalent to the code base's overall quality. A larger code base also

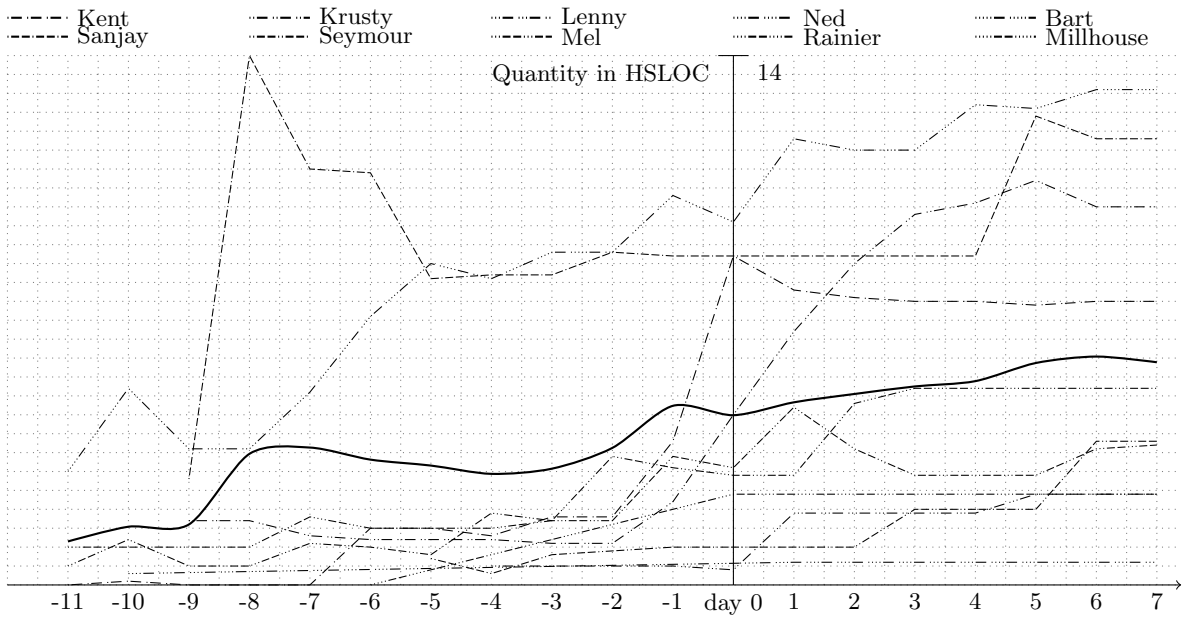

Figure 1: Amount of code contributed by the developers

leads to a stabilizing average score. The simple reason is here, too, that with more code a single change will have less effect on the average.

But the figure also shows that the intervention (starting at day 0) had only a small influence on code quality. The intended effect of increasing the code quality is not supported by our data. In fact, there is actually a small decrease in quality because day 0 coincided with a quality peak. We therefore conclude that the intervention did not have an adverse effect, too. It just did not show any significant effect. One could argue that quality scores are confounded by an end-of-project deadline. It may be that developers would normally have sacrificed quality in favor of functionality to complete all their tasks before the software is delivered. Our intervention could then have prevented that from happening. But this interpretation remains speculative.

3.3 Reputation scores and Subversion commit messages

We wanted to know if there is a correlation between the quality of a developer's Subversion commit messages and his quality reputation. The reasoning is that a developer who does write few documentation inside code will also neglect documentation in commit logs.

We went through all revisions created during the experiment and their according log messages. 471 revisions were created by the 10 developers who were part of the experiment (another ≈ 300 revisions were committed by other developers working on other sub-projects); some 60 revisions were not counted because they were submitted within seconds of an earlier commit with an identical log message, hinting that the developer just split his commit into parts. Each commit message was classified as bad (completely useless, e.g. empty, "commit", ...), ok (some effort made but not much information payload, e.g. "implemented the concepts"), or good. According to this classification, 342 commits were considered as good, 83 as ok, and 46 as bad.

We found a Spearman correlation of $r_s = 0.55$ between a developer's final quality score and the amount of non-bad commit messages among all his messages. It may be possible that not all developers were familiar with the commit message concept from the start of the project. Therefore we also looked at the messages that were created during the second half of the project only, when they had had time to learn the concept. Here the correlation is $r_s = 0.61$, which is statistically significant for $N = 10$ at $p < 0.05$. We infer that a developer's tendency to provide good commit messages hints at how likely he is to provide Javadoc in his code.

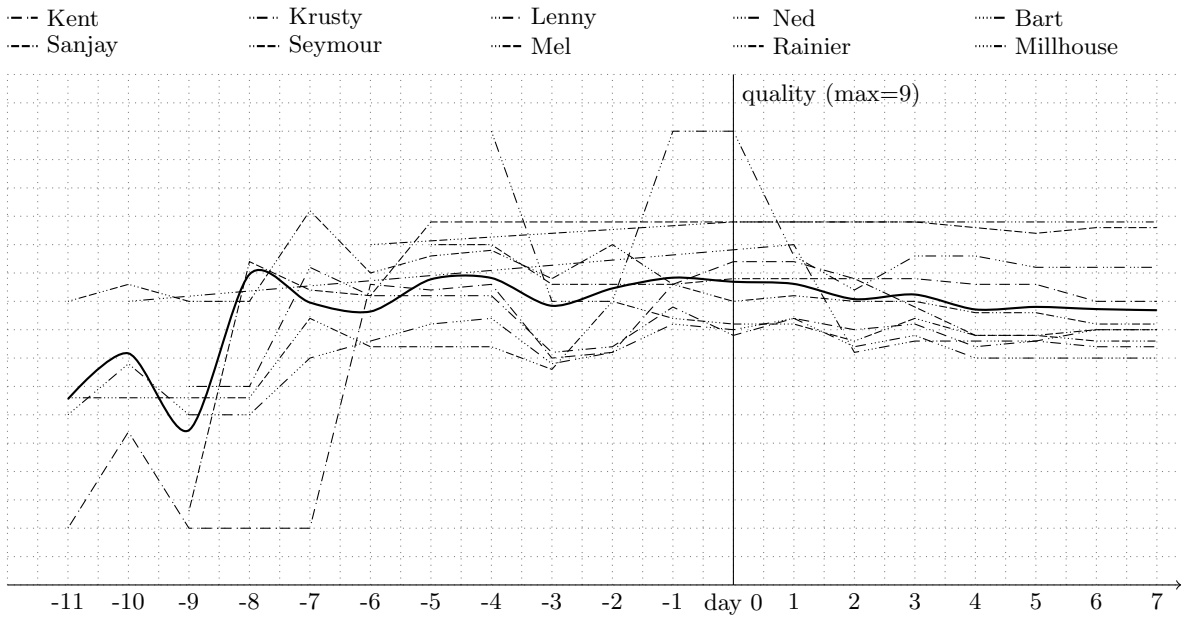

Figure 2: Average quality of authored code for each developer

3.4 Summary

We see that the amount of code authored by the individual developers increases during the project. At the same time, there is some fluctuation in their quality scores which stabilizes over time as more and more code is available. However, the quality graph does not reveal any significant change due to our intervention. It seems that the intervention was not appealing enough to alter developer behavior. In the next section, we investigate what the reasons for this ineffectiveness could be. We could show that quality reputation scores seem to be related to a developer's willingness to provide Subversion commit messages. As this is as expected, it indicates that our reputation scores are sound values.

4 Developer feedback

The purpose of the final interviews was to learn for the future design of similar interventions when it became foreseeable that the experiment would fail. We wanted to identify and understand critical aspects. The interview was based on a questionnaire with free text questions to qualitatively into reasons (see Section 4.1), and Likert items for quantitatively capturing the importance of certain factors (see Section 4.2). We investigated eight human "feelings" that we and colleagues deemed important to learn whether our intervention was

- present — How present were the reputation scores in your consciousness?
- fair — How fair were reputation scores suited to your work?
- important — How important were the reputation scores for you?
- understandable — How well did you understand the way reputation scores were computed?
- likable — How much did you like that reputation scores were computed at all?
- interesting — How interesting was it for you that there were reputation scores?
- fitting — How well do you think reputation scores fitted into the lab situation?
- acceptable — How acceptable are reputation scores for you?

4.1 Qualitative results

Some developers perceived the presence of reputation scores as high due to their competitive effect, and due to the daily emails sent out to developers. However, some developers noted that

this single email created ups and downs in presence. It could even happen that — because emails were sent in the evening — developers had forgotten about it until the next day. Another one ignored the emails because he did not consider his work (external API) documentation relevant!

Fairness was one of the more problematic aspects of our intervention. One developer stated that his score changed when he added a comment, and so he thinks it was fair. Others, however, criticized that the algorithm was strange and that they did not know how they could achieve good ranks. And it was mentioned that by contributing to files without writing comments but where others contributed them, one could gain reputation, it could create an impression of arbitrariness. Additionally, fairness was negatively affected by the pair programming because someone else might commit on one's account, and although two people are responsible for the code, only one is credited for it. The reality in the lab was not reflected very well, including the lab ideology that instead "the naming should be expressive" making description unnecessary.

Regarding importance, one problem was that reputation scores showed up late in the lab. Of course, the reason for this delay was the research methodology, but it made the scores come in as a surprise. Again one developer mentioned that for his kind of work, Javadoc was not important. So he considered the reputation scores as pointless and unimportant. Also reputation scores were perceived as being in conflict with "too much other stuff", and especially the priority "to implement a running application." But one developer noticed this conflict could be "just a lame excuse of not adding comments." Among the aspects that made reputation scores important were "for improvement of my work" and again the competition with other developers.

Understanding of the reputation scores was rated as the severest problem in the trial, and at the same time had a relatively high influence on the ranking. Only one developer thought that "the explanation was clear", but even he only expressed the hope that it was correct. Although developers "knew the idea of comments", it was the score that caused problems with understanding. More detailed information on the algorithm was missing and "the process of calculating the score was not clear" to some. One developer just "did not care about it" and so did not try to understand.

Developers liked the reputation comparably well, although there was some concern that it was not relevant. Other developers felt "bad to be second not first", "did not care about it", were not "following the score that keenly", or just had a diffuse dislike for it. Still the idea itself was not seen as a problem and liked "as a motivation for writing good documented code".

The perceived importance of reputation scores was negatively impacted by feelings of meaninglessness and priority conflicts. But the competition made reputation scores highly interesting.

The reputation concept was considered as fitting comparably poor into the evaluation environment. Although it was not precluded that it still "might fit", "applying it to our lab was not beneficial because of our lab nature." A major problem was pair programming because of group effects of pair work and the loss of contributor information in commits. Also Feature Driven Development was perceived as being in conflict with reputation scores. One developer mentioned that he noticed low acceptance among his peers which led to few commenting, and another said that he does not like the tendency in modern life to measure everything. However, others are "in agreement with what my score is & how it was computed", or found that it is a "nice idea as long as it is only for us, not grades/payment/other assessment of work done".

4.2 Quantitative results

Likert-scaled responses are presented in Table 1 (very low = 1, ..., very high = 5). The overall reception of our intervention reveals potential for improvement but also shows that it is not rejected outright. Fitting (2.44), fair (2.33) and especially understandable (2.11) are the most problematic areas, while likable (2.67) and present (2.89) are less problematic.

Unless stated otherwise, all correlation coefficients r_s denote Spearman rank correlations. Most correlations are not significant, which is probably due to the small sample size. That

	present	fair	important	under-standable	likable	interesting	fitting	acceptable	Average
Bart	very low	medium	medium	very low	medium	very low	very low	medium	2.0
Kent	low	low	high	low	low	medium	medium	low	2.5
Krusty	high	low	high	low	low	high	medium	medium	3.0
Lenny	high	medium	medium	low	medium	medium	low	medium	2.9
Mel	high	high	low	medium	medium	medium	low	high	3.1
Millhouse	medium	medium	low	medium	medium	medium	medium	medium	2.9
Ned	(no feedback received)								
Reinier	low	low	very low	very low	very low	very low	very low	very low	1.3
Sanjay	medium	very low	medium	very low	high	low	medium	very low	2.3
Seymour	medium	very low	very low	high	medium	medium	high	medium	2.8
Average	2.89	2.33	2.56	2.11	2.67	2.56	2.44	2.56	2.5
$r_s(mail)$	0.53	-0.05	-0.23	0.36	0.58	0.27	0.41	0.13	.35
$r_s(Q_l^{end})$	0.28	0.04	-0.01	0.28	0.49	0.18	0.38	-0.10	.25
$r_s(Q_l^{start})$	0.20	-0.01	-0.04	0.20	0.52	0.06	0.28	-0.15	.08
$r_s(Q_t^{end})$	-0.50	-0.42	0.44	-0.67	-0.03	-0.39	-0.11	-0.41	-.53
$r_s(Q_t^{start})$	-0.40	-0.14	0.41	-0.13	-0.37	0.01	-0.02	0.06	-.18

Table 1: Likert-scaled opinions about the intervention and correlation with score ranks

does not necessarily mean that detected correlations do not exist but the chances are non-negligible. As chances increase with lower coefficients, only correlations that have at least a medium strength (around $r \approx 0.3$ in the classification of de Vaus (2002)) are discussed below.

The average personal opinions correlate weakly with quality at the end of the evaluation period. The relationship with ranks published through email ($r_s(mail) = .35$) is a bit stronger. There is a chance that this increased strength might be attributed to the effect that profiteering leads to a better reception, or that a better overall reception of reputation leads to more investing in quality. Yet the difference is not significant and could be coincidence.

There is a strong relationship between published ranks and the perceived presence of reputation scores ($r_s(mail) = .53$). One explanation could be that those who feel that reputation has a high presence are more interested in achieving high reputation scores and ranks themselves, and therefore write higher quality code. But the following reasoning suggests that it is the other way around; that being higher in the ranks leads to a stronger perceived presence of reputation. If presence leads to investing in quality, regardless of the actually published scores, then the correct ranks Q_l^{end} at the end of the trial should show a similarly strong or even stronger correlation $r_s(Q_l^{end}) = 0.28 \not> 0.53$. Our data does not support this. Instead, future designs should take into account that having a low quality rank leads to low perceived presence of reputation. Cognitive dissonance theory can explain this observation[2] (Festinger, 1957).

The perception of fairness seems unrelated to actual reputation ranking. Similarly, we found no relationship for acceptability. Those who understood well how reputation scores were computed had a slightly better chance to achieve higher ranks ($r_s(mail) = .36$).

Developers who achieve high reputation ranks have a substantial tendency to like the computation of reputation scores ($r_s(mail) = .58$). It may be that profiteers plainly enjoy receiving the benefit of reputation (we describe above that we do not see such evidence). But it may also be that developers who care for code quality — and who therefore have high reputation scores — welcome any tool that supports code quality.

The factors that are most strongly correlating with high reputation ranks are the liking of having reputation computed, the presence of reputation, and the fitting of reputation into the XP environment. The only negatively correlated factor is the perceived importance of reputation.

[2] Humans want to always maintain a positive self-image. If they make observations, which are not congruent with positive self-image, then such observations are ignored.

For work quantity, most correlations are the inverse of the correlations for quality, and are in many cases stronger. We found a major negative relationship between quantity and average opinion $r_s(Q_t^{end}) = -0.53$. A simple explanation is the medium negative relationship between quality and quantity (see Section 2.4). More than that, mass contributors have a fairness problem, think that it is unfair ($r_s(Q_t^{end}) = -0.42$), have problems understanding how scores are computed ($r_s(Q_t^{end}) = -.67$), and do not feel that reputation is present ($r_s(Q_t^{end}) = -0.50$), acceptable ($r_s(Q_t^{end}) = -0.41$) or interesting ($r_s(Q_t^{end}) = -0.39$). But reputation scores often mattered to them ($r_s(Q_t^{end}) = 0.44$).

5 Threats to validity

Our field study included only a few developers, which makes correlation results hardly statistically significant. Statistical significance (5% threshold) for $N = 9$ is reached at $r \approx 0.6$. While some results are almost significant, most of them need to be treated with care. The reason why results are not statistically significant is probably due to the sample size, not because of the strength of the correlation. If the team had been bigger, more of our correlations would have been significant. Still, many results reveal a moderate strength, and the team size of about ten developers is the normal size for an agile software project or sub-project.

While it is true that the environment of the field experiment presented here is artificial to some degree, it is not set up for the experiment specifically but, as part of teaching activities, tries to resemble a natural setting as closely as possible. For example, while the subject pool features students, it is graduate students with development experience working in the domain of their education (Harrison and List, 2004). It is natural that experimental realism comes at the cost of experimental control.

Therefore some environmental influences were sub-optimal for our study. For instance the experiment might have been to short. Perhaps more time would have been needed for CollabReview to unfold its full effect. Yet we could not influence the length of the project. Another sub-optimal factor is that developers did pair programming. But these are the normal difficulties of experiments that are not conducted under fully controlled laboratory conditions. Researchers have to live with what is there. At the same time, this adds to the realism of field studies. To ensure that our results can be generalized to other software projects, more and larger studies in industrial software projects have to be conducted.

6 Related work

The question whether a correlation coefficient of $r = 0.5$ is weak, moderate or strong is difficult to answer. The answer is to some extent relative. When humans are involved, most outcomes have many causes so that no two variables alone are likely to be very strongly related. Here a correlation of 0.30 might be regarded as relatively strong. For describing correlation strengths involving human factors, this paper follows the recommendations of de Vaus (2002).

An experiment where CollabReview was successfully deployed to improve contribution to a work group's wiki is described in Dencheva et al. (2011). The main differences compared to the earlier study are the (i) domain (source code vs. wiki), (ii) social team structure (short-time lab vs. long-term collaboration), (iii) rewards (money vs. display), and (iv) pair development resulting in a loss of responsibility fidelity (pair vs. solitary contributing). Hoisl et al. (2007) present a similar study in a wiki using a different reputation system. Singer and Schneider (2012) gave points for committing frequently to a revision repository, emailed the scores in a weekly digest, and thereby could successfully alter developer behavior.

Checkstyle is a well-known tool for finding potential coding style problems in source code (Smart, 2008). Several authors have used Checkstyle and static analysis for grading and assessment of student programming assignments (e.g. Edwards, 2003; Smith, 2005; Loveland, 2009).

However, we do not know of instances where automated assessments are combined with fine-grained responsibility information to form reputation scores.

7 Lessons learned and future work

This section summarizes and recaps the lessons we learned from conducting the field experiment.

7.1 Lessons regarding the development environment

The presented experiment was the first field experiment with CollabReview in a programming environment. This encounter with reality has taught us several lessons. Firstly, CollabReview is not very well suited for pair programming because there is only one committer recorded in the revision control system while code is written by two developers. The social dynamics that are in effect between the changing pairs of developers, and which are not available to the reputation system, should not be underestimated. Also, the metrics were computed only for a part of the whole code base, which was assigned to the project. The analyzed code did not include Prolog and some Java parts. For example, it seems that Ned contributed only very little, which is probably wrong. This blurs contribution and responsibility mappings and probably leads to incorrect or at least skewed reputation scores. In turn, this reduces the understanding and perceptions of fairness in the team.

Extreme programming favors expressive naming of identifiers over comments. Much in line with this philosophy, project management had ambivalent opinions regarding comments and Javadoc during our field test. Some students perceived Javadoc as senseless. Management must clarify that there is no goal conflict, and that indeed naming and comments are important (cf. Raskin, 2005). Its full support for the employed measurements is necessary.

7.2 Lessons regarding the effectiveness of the reputation system

Classical management states that "you get what you measure". A metric might not measure the right things, leading to undesired behavior, but one should get what one measures (Hauser and Katz, 1998). Consequently, we should have gotten a lot of (potentially senseless) Javadoc comments. But we did not. The question is why?

Money seemed adequate for such a short, one-time project. Possibly winning the one prize was too risky or too much out of control for the individual (Hauser and Katz, 1998). Perhaps there should be more prizes. In previous work, we have been experimenting with other rewards as well (cf. Dencheva et al., 2011; Prause et al., 2010). For some developers, competition is an important motivator. For instance, Krusty mentioned competition as important factor and enjoyed it. According to our data, something with a high presence is needed. And in general, a better overall reception with regard to the different feelings leads to more investment in scores.

The reputation idea was mostly accepted. But developers expressed that acceptance depends on how scores are used. Especially developers who already care for quality, welcome reputation scores. However, there was no indication that it is profiteering which leads to a better reception.

Major contributors have a stronger influence on total code quality because they contribute more code. It is therefore of high importance to reach them with the intervention. But acceptance of the intervention among them is especially low. They might feel treated unfair because they give the software a lot of functionality, and might feel that this contribution is not valued enough. For a minor contributor (like Millhouse) it is much easier to achieve very high or very low reputation scores. It must be explained to major contributors that the purpose of the intervention is not assessing their performance. Instead, the goal is to improve the quality of the project, and here they have an even higher responsibility than minor contributors.

Low ranks result in higher perceived importance of reputation scores. This is a danger for team motivation! Even more so, as while being second is good, it "feels bad to be second not

first". Additionally, low ranks lead to low perceived presence, so especially poor-scoring developers need more presence of reputation and must understand that scores measure something that is important to them. Contrariwise, major contributors have the worst feelings about quality scores. But it is them who are very important because they influence the code base the most.

Create understanding of how scores are computed! Understanding has one of the highest correlations with the email rank. This is congruent with literature (cf. Hauser and Katz, 1998). Perhaps developers would just have needed more time to get acquainted to CollabReview. But also in the short time, the understanding of reputation could have been improved by providing more up-front training, and more immediate and elaborate feedback on developer actions. An alternative is to simplify the algorithm for measuring quality to make it easier to understand.

We are still convinced that reputation gaming will work when the experimental setup is amended in such a way that the relationship between investing in code quality and a desirable reward is firmer as, e.g. in the VIE theory by Vroom (1964); that means that

- developers need a better understanding of how to affect their score,
- responsibility assignments are less fuzzy by avoiding pair programming and scores are more under control of the individual (i.e. "Expectancy": effort in code ⇒ better score)
- a high score leads to a prize with a higher probability, e.g. by not only rewarding the first place (i.e. "Instrumentality": better score ⇒ prize), and
- the not necessarily monetary prize is worthwhile to achieve (i.e. "Valence", the prize).

8 Conclusion

We have conducted a field experiment with the CollabReview reputation system for source code in an agile software project. The goal was to affect ongoing development in such way that developers write more Javadoc. The reputation scores were not meant as a performance measure of a developer's productivity but as a compensation for the "endured pain" of doing what developers do not like to do: to write documentation (Selic, 2009).

The CollabReview reputation system was brought into the project after an initial phase where comparison data was collected. Developers started receiving a daily digest of their reputation scores, and a prize was announced for the winner. While the intervention probably had had some effect, no measurable quality improvement occurred. However, the experiment does not evidence that a reputation system is unsuitable for improving code quality. Instead, it shows that integration into development is difficult, and must be done right. CollabReview was previously deployed successfully in a wiki, so it was a surprise that this experiment "failed".

The major contribution of our work is therefore the lessons that we learned: Measurements must be implemented carefully to measure the right things, and to not endanger team spirit. A low rank leads to low perceived presence but high importance, i.e. weak positive but strong negative effects. Already the second place leads to a bad perception. Furthermore, developers who care for quality welcome a tool like CollabReview that supports them. Yet there is no indication that it is the profiteering that leads to a better overall reception. But a better overall perception leads to more investing in quality. Of all factors, understanding seems to be the most essential factor for performance improvement. Especially mass contributors had bad feelings about scores but they mattered to them the most, creating feelings of unfairness. However, there was few rejection and acceptance mostly depends on how scores are used.

With our work we guide the design of similar systems, and further the understanding of the social dynamics that reputation systems cause in software projects. We also found that developers with high reputation scores (resulting from Javadoc) are likely to write commit messages. The reason is perhaps a documentation-affirming character trait. This finding suggests that reputation scores do not capture an arbitrary value but are a sound measure.

In the future, we want to repeat the field experiment in a more suitable environment without pair programming, and carefully consider the lessons learned. Ideally, the later experiment has

more participants, and can run for a longer duration. At the same time, we want to write a daily log of project events to be able to relate events to observations of reputation scores.

Acknowledgments

We are grateful to the teachers of the XP Lab 2011 — Armin B. Cremers, Daniel Speicher, Paul Imhoff — who kindly allowed us to conduct our studies, and the participants for being our guinea pigs. Furthermore, we like to thank Markus Eisenhauer, Uwe Kirschenmann, René Reiners, Gabriel Bonerewitz and the PPIG reviewers for their advice on the experimental setup, designing the interview, their opinions on the obtained results, and the valuable hints for improvement.

Bibliography

de Vaus, D. A. (2002). *Surveys in Social Research.* Routledge, fifth edition.

Dencheva, S., Prause, C. R., and Prinz, W. (2011). Dynamic self-moderation in a corporate wiki to improve participation and contribution quality. ECSCW, pages 1–20. Springer.

Dubochet, G. (2009). Computer code as a medium for human communication: Are programming languages improving? In 21^{st} *Annual Workshop of PPIG*, PPIG.

Edwards, S. H. (2003). Teaching software testing: Automatic grading meets test-first coding. In *OOPLSA Companion*, pages 318–319. ACM.

Festinger, L. (1957). *A Theory of Cognitive Dissonance.* Stanford University Press.

Harrison, G. W. and List, J. A. (2004). Field experiments. *J. of Econ. Lit.*, XLII:1009–1055.

Hauser, J. and Katz, G. (1998). Metrics: You are what you measure! *EMJ*, 16(5):517–528.

Hoisl, B., Aigner, W., and Miksch, S. (2007). Social rewarding in wiki systems — motivating the community. In *Online Communities and Social Computing.* Springer.

ISO 9126-1 (2001). ISO 9126-1: Software engineering - product quality: Part 1: Quality model.

Jøsang, A., Ismail, R., and Boyd, C. (2007). A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644.

King, P., Naughton, DeMoney, Kanerva, Walrath, Hommel, et al. (1997). Java code conventions.

Loveland, S. (2009). Using open source tools to prevent write-only code. In *Conference on Information Technology: New Generations.* IEEE CS.

Prause, C. R. (2011). Reputation-based self-management of software process artifact quality in consortium research projects. In *ESEC/FSE*, pages 380–384. ACM Press.

Prause, C. R. and Apelt, S. (2008). An approach for continuous inspection of source code. In 6^{th} *International Workshop on Software quality*, WoSQ, New York, NY, USA. ACM.

Prause, C. R. and Durdik, Z. (2012). Architectural design and documentation: Waste in agile development? In *International Conference on Software and System Process.* IEEE CS.

Prause, C. R. and Eisenhauer, M. (2012). First results from an investigation into the validity of developer reputation derived from wiki articles and source code. In *CHASE.* ACM.

Prause, C. R., Reiners, R., Dencheva, S., and Zimmermann, A. (2010). Incentives for maintaining high-quality source code. In *Psychology of Programming Interest Group Work-in-Progress.*

Raskin, J. (2005). Comments are more important than code. *ACM Queue*, 3(2):64–62 (sic!).

Seibel, P. (2009). *Coders at Work: Reflections on the Craft of Programming.* Apress.

Selic, B. (2009). Agile documentation, anyone? *IEEE Software*, 26(6):11–12.

Singer, L. and Schneider, K. (2012). It was a bit of a race: Gamification of version control. In 2^{nd} *International Workshop on Games and Software Engineering.*

Smart, J. F. (2008). *Java Power Tools.* O'Reilly Media, first edition.

Smith, D. (2005). Gide: An integrated development environment focusing on agile programming methodologies and student feedback. In *WCCCE.*

Spinellis, D. (2011). elyts edoc. *IEEE Software*, 28:104–103.

Vroom, V. H. (1964). *Work and Motivation.* Wiley.