Gamification for Enforcing Coding Conventions

Christian R. Prause
DLR Space Administration
Königswinterer Str. 522-524
Bonn, Germany
christian.prause@dlr.de

Matthias Jarke
RWTH Aachen
Institut i5
Aachen, Germany
jarke@informatik.rwth-aachen.de

ABSTRACT

Software is a knowledge intensive product, which can only evolve if there is effective and efficient information exchange between developers. Complying to coding conventions improves information exchange by improving the readability of source code. However, without some form of enforcement, compliance to coding conventions is limited. We look at the problem of information exchange in code and propose gamification as a way to motivate developers to invest in compliance. Our concept consists of a technical prototype and its integration into a Scrum environment. By means of two experiments with agile software teams and subsequent surveys, we show that gamification can effectively improve adherence to coding conventions.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.9 [**Software Engineering**]: Management

General Terms

Human Factors, Documentation, Management, Measurement

Keywords

code style, gamification, software quality, experiment

1. INTRODUCTION

The standard ISO-9126 establishes a basic model of software quality based on six quality characteristics (e.g., functionality, maintainability) each subdivided into several subcharacteristics (e.g., analysability, changeability). These characteristics can be roughly divided into characteristics of external and internal quality. The famous iceberg metaphor illustrates this relationship: external quality is the iceberg's surfaced part that can be easily seen by users of the software. Internal quality is the part dangerously hidden below the surface. Among the submerged parts, the *maintainability* quality characteristic means how cost-effectively developers can continuously improve and evolve the software.

Maintainability is broken down into the sub-characteristics analyzability, changeability, stability, testability and maintainability compliance. They describe how easy places to be changed and causes of faults can be located, how well future changes are supported and can be realized, how much the software avoids unexpected effects from changes, how well the software supports validation efforts, and how compliant the interior of the software is to maintainability standards and conventions. Maintainability is like an internal version of the usability quality characteristic, i.e., how easy developers can understand whether source code is suitable for certain purposes and how to use it for particular tasks and in specific conditions, how well inner workings can be learned, how much developers stay in control of the changes they make, how attractive the inner workings of the software appear to the developers, and how compliant they are to standards, conventions, style guides or regulations. When we speak of understandability, learning, attractiveness, and compliance in this paper, we mean this internal view and not the view from the end-user's external usability perspective.

Maintainability is important for the evolvability of software. We argue that it is related to knowledge, and information exchange in the team. As a hidden quality characteristic, however, it is easily overlooked, and requires efforts and discipline that are not granted. In particular, human behavior is controlled by trading off the valences[1] of alternative actions all the time. If the valence of complying to conventions is not high enough, developers will easily be drawn to other activities. The dilemma of information exchange commences. The more freedom developers have, the more important such valence aspects are. Internal un-quality, also known as technical debt, increases (see Section 2).

We propose gamification as a solution. *Gamification* refers to the use of design elements characteristic for games in non-game contexts. It has gained considerable attention in domains like productivity, finance, health, education, news, and media. In this sense, gamification does not mean "play" but means to optimize one's behavior with respect to the rules in the sense of mechanism design [18, 55]. In a real-world context (programming) with real-world effects (readability of code and rewards), gamification is what connects the real-world on both sides using mechanisms known from games. For this approach to work, two different integra-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786806

[1] In psychology, the term *valence* means the intrinsic attractiveness or aversiveness (i.e., positive or negative valence, respectively) of an event, object, or situation [22].

tions must be achieved: Firstly, a technical solution must be integrated into the development environment so that the necessary development data can be obtained and prepared for gamification. For every developer, a reputation score is calculated that averages the compliance to coding conventions of the files the developer contributed to. Secondly, the gamification must be integrated socially to affect the developers' valence judgments so that desirable behaviors will emerge more often. We rely on the typical game design elements of personal scores, transparent rules, leader board, and real-world individual and group incentives that can be earned as team (see Section 3).

In order to evaluate our game design, we set up two parallel experiments with software development teams following the agile Scrum methodology. We distributed postgraduate developers with similar skills onto two teams of greenfield projects. Both teams would run through an intervention and a control phase resulting in experimental between-group and within-group designs. Surveys using questionnaires were scheduled with each team at the end of their intervention period to capture qualitative and quantitative information about the gamification. Furthermore, participation in customer meetings and feedback from the project leaders enrich our perception of what happened during the experiments (see Section 4).

Evaluation of the experiments shows a clear effect of the interventions in both teams. Both teams invested the efforts necessary to achieve the designated convention compliance. The game elements had the desired effects, and feedback from the developers suggests that the investments in convention compliance were worth the efforts (see Section 5; threats to validity are discussed in Section 6).

Coding conventions are widely recognized as a means to improve the internal quality of software. Much work has been put into conventions, commercial tools relating convention compliance and technical debt, and the enforcement of conventions with techniques like reviews or strict tool-based enforcement. We compare our work to such approaches. The advantages of our approach are relatively comparably low efforts in personnel, and higher flexibility on the developers' side. Furthermore, other research successfully applied gamification to shape developer behavior (see Section 7).

Our contributions lead us to conclude that gamification works to improve coding convention compliance and is worth further research (see Section 8).

2. CODING CONVENTIONS

Software is a highly knowledge intensive product. Its source code results from a non-algorithmic, experimental activity of discovery and invention. Developers identify issues, experiment with possible solutions, and come to decisions in a process involving justifications, alternatives and trade-offs [9]. Its source code is "the only precise description of the behaviour of the system" [24], and it must be both, executable by machines and understandable to humans. A developer must first understand the code before being able to successfully implement changes. Code therefore plays a key role in the information exchange between developers and for preserving knowledge. If the necessary knowledge is not preserved well and easy to retrieve for a human, i.e. readable, software becomes costly to maintain and evolve [14]. When internal quality decreases, efforts in re-work increase (called "un-quality" by Philip B. Crosby), and technical debt grows into a danger to the project [13]. Some therefore even call not putting efforts into such information exchange unprofessional "anti-team behaviour" [33]. If information exchange is so important, why are programmers "notorious" [44] for their dislike of writing down knowledge in a readable form [26]?

The literature on programming lists countless different motives: time and schedule pressure; missing responsibility, education and professionalism; lack of incentives; carelessness, incompetence, "sapient" developers; human nature; additional (cognitive) effort; or operand conditioning, perceived boredom and pathological gambling. Writing down information in a readable way costs a developer his precious time and (cognitive) efforts but has little immediate value for himself. The potential values pay off much later and mainly benefit others [11, 15, 23, 30, 32, 33, 36, 41, 43, 48, 49, 56].

We note that information exchange through code is an instance of the more general *dilemma of information exchange* (cf. [16]): A developer must constantly chose between implementing new functionality, and spending efforts on information exchange. Writing helpful source code comments and thinking about the effect for readability of every single keystroke does not work on autopilot [48]. Developers subconsciously weigh up motives like time pressure, cognitive effort, and joy of creating new functionality against other motives like the desire to conform to management expectations, pro-social information exchange and the satisfaction of achieving quality. According to psychological theory, factors weighted by personal expectations and beliefs, sum up on the pro and con sides of the decision. The developer will decide for the alternative with the greatest expected *valence*.

Developers will disregard information exchange, when its expected valence is lower than that of other activities. A social dilemma then commences, if it is still valuable to the team as a whole. And indeed, the value for the sending developer is lower than for his audience: A developer who wrote the code himself, will have a deep understanding of its functions, inner workings, side effects, and constraints. He will not need to read that knowledge from the code; at least not in the near future while his memory has not faded and the code remains unchanged (cf. [38]). However, other developers miss that knowledge. For example, they reinvent the wheel over and over again [28], wander like explorers of a new world looking for ways to solve problems [35], and cannot use their advanced skills [51]. They have to learn the code first to successfully implement changes.

Readability is a judgment of how easy source code is to understand, i.e., how easy the knowledge encoded in it can be retrieved by a human [14]. Coding conventions support information exchange by improving understandability and readability [50]. They facilitate communication and collaboration between developers by maintaining uniformity [31]. Nordberg compares conventions to homeowners' associations that support quality by reducing individuality. They prevent code from getting unreadable due to mixtures of unconventional styles [36] and therefore help to reduce what Graham [23] calls the common-room effect, which makes code written by different developers feel bleak and abandoned, and accumulate cruft.

The benefit of coding conventions often is not so much that each rule is well-grounded. In fact, many conventions implement common wisdom, memes of a project's coding culture, and subjective opinions about aesthetics [36]. Take Linus Torvalds' famous comment as an example: "First off,

I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture." From an objective point of view, many rules might not stand scientific scrutiny. Their benefit is that they reduce style deviations, which are nothing but distracting noise when reading code [50]. Despite a human language text remaining understandable in principle, who would seriously doubt the benefits of orthographic and typographic conventions like spelling, hyphenation, capitalization, word breaks, emphasis, punctuation, line length or spacing? Conventions are low-hanging fruits that are an important and pragmatic ingredient to making software better, which is supported by broad interest in the topic (see Section 7). We are therefore convinced that they are relevant for internal quality.

3. GAMIFYING COMPLIANCE

Gamification is related to mechanism design. It uses game design elements for a not-playful purpose in a non-game context. It does not aim to create a full-fledged game for enjoyment. Instead, it employs design elements that are known to work from games to set up structured rules for a competitive strife toward goals for its players [18]. Embedded in a real-world setting, the additional rules increase the valence of putting effort in desirable behaviors. As a result, developers will more often choose to comply to coding rules. Our gamification of coding conventions consists of two parts:

1. a technical component (the CollabReview prototype) that integrates into the software development process to mine the necessary data, (Section 3.1) and
2. game elements that have valence for developers and motivate them to invest in complying to coding conventions (Section 3.2).

3.1 Technical Component: Obtaining Data

At the heart of the technical component is a *reputation system*. It gathers reputation statements by observing users' actions and collecting feedback. The statements are then integrated into a model of the developers' reputation, expressed as personal scores [21].

The personal reputation score reflects the average compliance of a developer's code with the coding conventions, ranging from +10 (fully compliant) to -10 (extremely deviant). For determining it, the reputation system must

1. determine compliance of each source file (Section 3.1.1),
2. assess who is responsible for each file (Section 3.1.2),
3. integrate both into the reputation score (Section 3.1.3).

Note, that contribution quantity is not represented in the reputation score. The rationale is that we do not want to measure performance but promote compliance.

3.1.1 Determining Compliance

Checkstyle is an open source static analysis tool for Java. It reports different rule violations like formatting, Javadoc-related, magic numbers, and many more using four severity levels: ignore, info, warning and error. It is highly configurable and can be made to support almost any coding standard. The Sun Code Conventions and Google Java Style are supported natively [1]. We automatically compute the compliance judgment $-10 \leq q(f) \leq 10$ for each file f as the density of reported violations, i.e., the number of violations v_c in the top three severity levels divided by the number of non-empty lines c_{sloc}:

$$q(f) = \max\left(10 - \frac{10 \times v}{c_{sloc}}, -10\right)$$

3.1.2 Assessing Responsibility

The developer who added or last modified a line of code owns it and is responsible for it. Using an algorithm similar to Subversion's `blame` command, responsibility is mined from the code's revision repository. The idea is that any new version of a file is created by manipulating the contents of its *parent*. Reconstructing these manipulations means solving the string editing problem. It results in an edit script with the shortest series of changes (insertion, deletions and substitutions) that transform the parent into the new version.

Our algorithm differs in three aspects from Subversion: First, it is indifferent to white space changes. Second, clone detection finds if a code block is moved unchanged within or between files to retain original authorship. Third, the parent of a new version is any file or version that is most similar to it, which is not necessarily the direct previous version of the same file. The rationale behind all modifications is to retain responsibility of contributors even though others may make minor changes to the same code [40].

The results of this process are responsibility ratios $0 \leq r(d, f) \leq 1$, where $r(d, f)$ is the responsibility of developer d for file f. $r(d, f)$ is proportional to the number of characters in lines owned by d in f. For any given file $\sum r(d, f) = 1$, regardless of its size.

3.1.3 Computing Reputation Scores

On average, a developer should have a reputation score that approximates her average compliance of contributions. For reasons of fairness, a file to which she has contributed more (or which is large) should have more influence on the score than a file for which she has fewer responsibility (or which is smaller). Reputation $k(d)$ is defined as the average of the compliance of code $q(f)$ over all files F in the project, weighted by size $w(f)$ and responsibility of the respective developer $r(d, f)$:

$$k(d) = \frac{\sum_{f \in F} q(f) r(d, f) w(f)}{\sum_{f \in F} r(d, f) w(f)}$$

The source code of the CollabReview platform is available as open source (see [3]).

3.2 Social Component: Valence and Influence

Reputation scores are an essential component to the gamification. Yet their mere existence does not affect developers because it has no valence for them. Research shows that users will value reputation quite exactly as much as it provides valence [29]. The elements of gamification are then what presents valence to users and lets them pursue higher reputation scores. Common examples of such elements are leader boards, levels or badges [18]. The choice of gamification elements must be calibrated to the specifics of the environment (cf. [42]). Farmer and Glass give a guide to designing gamification elements using reputation systems [21].

When we created the design, we had knowledge from previous experiments and studies. We had thoroughly studied theoretical work, conducted expert workshops, and interviewed software engineering experts to identify potential problems. Additionally, we had tested the technical aspects of reputation computations to gain confidence in their correctness. In general, it is important to have a good understanding of the users and their organizational environment.

3.2.1 Personal Scores

Personal scores are the technical basis of most gamification designs. Choosing an environment where score can be obtained is therefore essential. For example, responsibility can be difficult to determine in pair programming situations when two developers work together but only one commits to the repository.

Presenting individual scores let team members develop a feeling of self-efficacy when they see how their actions affect their score. It helps them to learn and optimize their behavior and understand the game. We therefore disclosed the personal scores to the developers.

3.2.2 Transparent Rules

In our experiment, developers would perform distributed development, mostly working off-site. Such setting hinder informal interaction, information exchange, and the establishing of resilient social structures. Consequently, instructions regarding the game — like briefings, goals and reputation reports — would have to be complete, precise and self-explanatory. A gamification can easily fail due to a lack of understanding of how to "play".

Furthermore, communication of reputation scores is a critical ingredient. Scores must be communicated in a way that developers take note of them, easily understand them and can relate them to their actions. An important element hence are intermediate reputation reports sent to the developers via email. Each report explains how scores are computed. It contains the relevant computation basis of compliance of each file $q(f)$ (including the reported violations) and the two main contributors with the highest responsibility $r(d, f)$ for the file.

While the reports were generated automatically, we sent them manually as emails to the team's focal point, who was then responsible for distributing it further in the team. This manual process increases the visibility and perceived importance of the reports. For the same reason, we kept the total number of reports sent below ten.

3.2.3 Leader Board

In a survey we conducted earlier, the typical elements used in gamifications (levels, badges, leader boards) were deemed to be ineffective and have only limited valence. A problem with leader boards, in particular, is that they emphasize the top ranks, while having few motivational influence on the vast majority of the remaining ranks in the middle. An exception are leader boards with negative reputation that were strongly unpopular. For example, *technical debt* is valuable as means of conveying overall information about internal quality. However, it is not suitable as reputation, i.e., everybody would have his own share of technical debt.

The compliance reports still contained a table of reputation scores. However, it was not intended to function as a leader board. Its purpose was to inform the team of its overall progress and to create transparency within the team. It shows developers their own and the others' scores to support self-coordination of group work, who needs support, and what has to be done to achieve the rewards.

3.2.4 Individual and Group Incentives

The main driving force of the gamification is a mixed individual and group incentive that can be earned either by an individual or by the whole team. Group incentives have the advantage over individual incentives that they have stronger effect because members attempt to manage one another through peer pressure and other social sanctions. At the same time, this can lead to friction in the team. Adding individual incentives reduces this risk [25]. Furthermore, group rewards allow developers to help each other.

The group incentive was announced to be given to the whole team if each team member achieved at least a score of $k(d) \geq 9$ (which is no more than one violation per 10 lines of code) by the end date. While the condition took into account the score of each developer individually, developers could help one another by fixing violations in someone else's files. Still, personal scores allowed the teams to manage conformance. The target threshold of 9 out of 10 possible reputation points was chosen not to be too easy but also not extremely difficult to achieve.

For the case the group incentive was completely out of reach due to too many deniers, an individual incentive was also announced. This individual incentive would only be given to developers, if the team failed to achieve the team goal. The purpose of the individual incentive was as a fallback and to reduce friction in the team if several developers would not consider the prize worthwhile achieving. In that case, the individual prize was announced to developers with the highest reputation scores at the end of the experiment. On the one hand, at least developers interested in the prize would be able to achieve it. On the other hand, a team of deniers could not decide as a whole to not go for the prize because the highest scoring developers would still get it.

An effective reward must be valuable and seem attainable. In an earlier experiment, we offered a prize money of several dozen Euros in a student lab for ranking first. As only one developer could earn the prize, the expectancy-value was perceived as too low, and consequently the reward turned out not to be attractive. We therefore offered the prize to the second ranked as well. Instead of money, we offered a bonus of one third of a grade on the final project grade. No reward was offered for the third rank because three willing developers were about one third of the members of a team. We considered three developers as a critical mass that would be strong enough to push the whole team towards the group goal situation instead.

4. SETUP OF THE EXPERIMENT

We aimed our experiment at the gold standard of experimentation: Out of two groups, one group receives the intervention while the other one does not. Ideally, any observed difference can then be attributed to the intervention. The two questions that drive our experiment are:

Principal Question: Does gamification of coding convention enforcement lead to improved compliance?

The results of a rigorous experiment alone should provide sufficient evidence for the above question. However, surveying the test subjects after the experiment can increase

confidence in the results by revealing processes and motives that were in effect. Therefore, it is valuable to investigate how the gamification intervention itself is perceived by the subjects. For example, if the gamification is present in the subjects' minds during the experiment and is considered important, then it is quite probable that it also influences their behavior. Qualitative investigations on these motives can provide hints for improvement and further explanations.

Secondary question: Does gamification cause undesirable social effects?

Another important aspect is that gamification should not have negative side effects on the team itself. For example, group incentives can cause friction [25], or the gamification could over-emphasize convention compliance. For investigating the second question, we analyze survey responses and have course instructors report suspicious incidents.

4.1 Frame Conditions

As part of the teaching activities of a local post-graduate teaching institution, two independent software teams executed separate projects following the agile *Scrum* methodology. The teams consisted of rather experienced developers, however, included also ones with only some experience (see Section 4.2). To avoid potential problems with legacy code, both projects were green-field projects. It should be noted that the purpose of the projects was not the experiment but that they were supposed to deliver actual functionality/software. The gamification experiment was only a piggyback. Therefore both projects had different functional topics, while still being setup very similar from a developmental point of view.

The duration of the projects was more than two months, including 47 work days (excluding weekends and several public holidays). Each team was responsible for organizing itself, e.g., resource allocation or scheduling. However, one Scrum sprint retrospective and a review meeting with the instructors acting as customers had to take place at the end of each week. The course instructors and the experimenter were different persons. The experimenter only appeared as a guest at the respective intervention start and ending meetings.

The size and organization of the projects allowed for an experiment with both *within-group* and *between-group* designs. Team A starts its project while receiving the CollabReview intervention. During this time, the performance of Team A is compared to the performance of Team B (between-group). After about half of the project duration, the intervention ends for Team A and begins for Team B. At the end, the performance of Team B is compared to its prior performance without the intervention (within-group). The performance of both teams is monitored all the time.

4.2 Distribution onto Teams

For a successful between-group experiment, both groups need to be as similar as possible. We manually distributed the developers onto the two projects in advance.

In the end, the two teams consisted of nine and eight developers, respectively. Additionally, developers were distributed in such a way that different skill levels were evenly distributed. We determined skill levels by means of a short questionnaire that developers filled out. The questionnaire asked them to assess their Java coding skills, other coding skills, experience in Scrum, knowledge of software engineering processes, and practice with collaboration tools like Subversion. The collected self-assessments were combined into a single overall average skill score. (The combined per-developer score has been included for reference in the final results Table 2. Three developers joined Team A last-minute, so no self-assessment data is available.) In the end, average skill scores between teams differed by less than five percent. On these scales, the developers considered themselves as rather experienced in programming in general and with Java, and were knowledgeable of the software development process. They had some experience with collaboration tools but few with Scrum.

4.3 Experiment Time Schedule

The time for the end of the intervention for Team A and start for Team B was fixed to be at the half of the project duration, i.e., after about one month, i.e., 22 work days excluding weekends and public holidays. Development in both teams started after about two weeks because earlier software development life cycles phases (e.g., requirements elicitation) had to deliver early results first. While the intervention had been announced to Team A on the first day, the first compliance report was sent to Team A only after two weeks when development was really ongoing (Day 10). The intervention ended for Team A, and started for Team B on Day 23.

4.4 Intervention and Briefing

Each intervention started with an instructional email to the respective team's *Scrum Master*, who was responsible for circulating it within his team. The instructional emails described the gamification and contained the first compliance report. It displayed the developers' current reputation score, and details on responsibilities and detected violations as hints for improving reputation.

The teams were also informed that any intermediate reports sent to them were not important for the reward. Only the final score at the end of the respective intervention period was relevant. That meant that both teams should have enough time to carefully approach a better score. Team B received the explicit advice to carefully improve style and fix problems over time, and not to heedlessly fix them immediately, or hope to fix them in the last few days.

The teams also received the following additional hints:

- If developers liked, then they could use the static analysis tool themselves to check their code before check-in.
- Most IDEs like Eclipse are capable of automatically fixing style issues, which should easily let a developer get rid of several violations without manual effort.
- While every developer had his own score, they could help each other by fixing someone else's code. They could win individually or as a team.

4.4.1 Surveys after the Intervention

At the end of each team's intervention, a questionnaire was given to the developers to gain a better understanding of how CollabReview was perceived. We wanted to learn for future designs. Each question contained a 5-point Likert scale ("How much...") and a free text item ("Why was it...") for capturing feedback both quantitatively and qualitatively.

It asked how *present* scores were in the developer's consciousness, how *important* the gamification was for them,

how *fair* it was, how well the developer did *understand* its computation, how much the developer did *like* that reputation was computed, how well it was *fitting* into the development process, how *acceptable* the gamification was, how much *influence* it had on the developer's way of programming, how much *additional effort* it caused for the developer, how much the code *readability improved* due to the intervention, if it would have made a *difference* for the developer to only see a team's average score instead of individual scores, and if the developer had privacy *concerns*. (The identifiers in *italics* re-appear in the next section.)

5. RESULTS AND EVALUATION

This section presents collected reputation data, reflects observations made during the experiment, reproduces results from the final questionnaire, and finally interprets and summarizes the results.

5.1 Collected Reputation Scores

Figure 1 shows how reputation developed over the course of the experiment. The continuous lines are the average reputation scores of Team A in red and Team B in blue. The "average" reputation is weighted by each developer's level of contribution to the project. It has the same value as if all code had been written by a single developer. In addition, the figure depicts all developers' individual reputation scores with dotted lines in the team's color.

Four events are explicitly marked with thick gray lines. These are the start and end dates of the intervention of Team A and Team B, respectively. Team A received their first compliance report (and the experimental instructions) at the tenth day when their code base first contained more than 200 lines of code.

The average reputation of Team A does not change very much during the first two weeks of their intervention. Instead, an average reputation of about 5 seems to be achieved naturally without much attention. The reputation of Team A then suddenly goes up to the maximum possible score (10) in the last few days before End A. From then on, after the intervention, average reputation slowly begins to decrease.

As opposed to this, Team B starts out with an average reputation of 5. During the next weeks, reputation slightly lowers until reaching a "natural" 3 at the day of Start B. For two and a half weeks, Team B steadily works its way towards the team reward threshold of 9. They then use the remainder of their intervention period (End B) to further improve their scores a bit while never reaching the maximum of 10.

Both teams reached the average reputation needed to be awarded the team reward at the end of their intervention period. While Team A seems to have been more deadline oriented, Team B slowly worked toward the goal and then maintained a safe margin over the threshold.

Translated into violation density, the reputation scores mean: At the start of their respective interventions, Team A had ≈ 0.65 violations per line of code, while Team B had ≈ 0.75. Until the end of their interventions, Team A improved compliance to zero violations per line of code, while Team B improved compliance to almost zero violations.

5.2 Observations

This subsection reflects on a few observations that we made during the experiment. As teams were distributed and conduction of the work was in their own responsibility, observations were restricted to the weekly Scrum review meetings. However, a few anecdotes emerged:

Developer A3 misread the briefing of the intervention, when reputation scores were first revealed to his team. He reported spending the whole night trying to fix problems but finally gave up as he saw no chance of reaching the target score. In the Scrum review, he was frustrated and expressed feeling offended because he had worked so much and now feared he would not get the prize. But his team could appease him as he misread the briefing and still had enough time left to fix his reputation score.

During the next weeks, however, A3 did not improve his score very much. Instead, his team helped him fix problems with his code only the day before the deadline. In the Scrum retrospective at the end of the intervention he argued that the conventions were driving him mad as they were taking him too much time. He continued that it was only for his team members who saved him. Interestingly, he followed up on his original statement several weeks later when the intervention of Team A had long ended. He said that he had actually learned to appreciate the better readability resulting from convention compliance and, to his dismay, had noticed its decay after the intervention had ended.

B7 also stated that he had someone else improve the code for him. Indeed, it seemed to be a strategy in both teams to have certain members specialize in fixing compliance when the deadline approached. CollabReview leaves this flexibility to developers while at the same time encouraging them to efficiently and autonomously negotiate improvement work.

The members of Team A increased their scores overhastily right before their intervention ended. This could be the reason why they went for a perfect 10 instead of the necessary 9: They had no time left and had to get it right with one shot; so they did not risk a single violation. A problem with this approach was that they broke their software right before presentation at the Scrum review. Some functionality got lost in the huddles.

5.3 Questionnaire Feedback

Figure 2 summarizes opinions about the intervention. As the items *additional effort* and *concern* capture negative opinions, the responses shown in the table were inverted for easier visual comparison. Row averages (in parenthesis) are presented as an indication of "overall feeling" that should be treated with care. On average, Team B (3.9) perceived the intervention better than Team A (3.2).

Present: The valence of the team prize was a major reason why the gamification had a high presence for developers. But developers also said that they cared for the readability of their source code and found the intervention helpful, and that they wanted to have a high score. Some developers of Team A admitted to have started caring more for the compliance of their code when approaching the deadline. The idea of paying some attention all the time was more widespread in Team B.

Fair: The gamification was perceived as quite fair. The biggest concern was about the computation of scores. Developers with many contributions could (in absolute numbers) catch more violations, and thus have a lower score. Vice versa, fewer contributions meant less potential convention violations. These developers considered neglecting of quantity demotivating. For some developers, a concern was that Checkstyle did not support aspects like identifier names

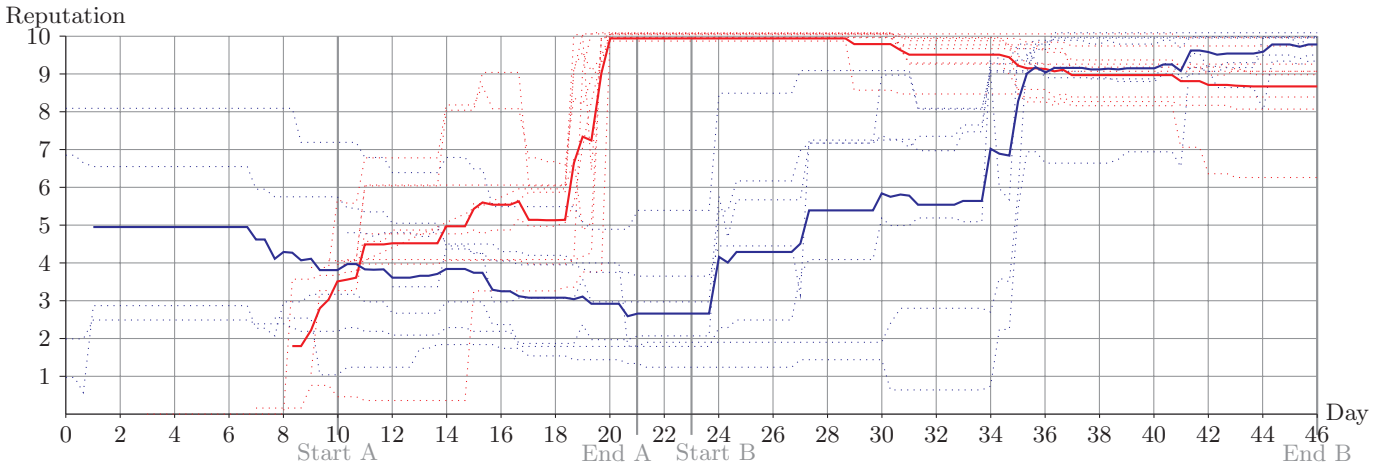

Figure 1: Average team scores (Team A red, Team B blue) and scores of individual team members (corresponding colors, dotted)

while being too picky about aspects like trailing spaces. Indeed, this is a valid remark regarding Checkstyle; yet not so much of the enforcement. Other developers, however, found that Checkstyle was making fair judgments, and that scores were helpful and gave them the feeling to have achieved something.

Important: The gamification was important to the majority of developers. Reasons why they found it important were the prizes, wanting to finish on a top rank in the leader board, the desire write clean code, and as a way to reflect on one's own work. Those who found it less important, said that they preferred to focus on functionality.

Understand: The majority of developers felt that they understood how reputation was computed. A bit of experimenting and studying the reports was still necessary to understand the score and how to influence it. This aspect was particularly relevant for Team A because they started to take care of the scores rather lately.

Like: In Team A, the gamification caused some distress for functionality-oriented developers due to their needing of help from others. Developers in Team B generally liked the intervention from an educational point of view and because they were kept alert about readability. They had no feeling of a contest but felt they were collaborating nicely. They particularly enjoyed that all team members were encouraged to comply to conventions.

Fitting: In general, CollabReview was perceived as fitting quite well into the development environment. However, developers noticed that the high pressure coming from the Scrum environment was further increased by one more aspect that they had to take care of. A recurring topic was that the compliance aspect was perceived as distracting (cf. [48]). In particular, developers wanting to focus more on functionality felt they were forced to invest in compliance as well. One developer criticized that in group work individual scores would not fit well. However, proponents held up that clear code benefited the team as it was easier to read for all others, and that reputation scores were just the right motivation. Scores allowed them to see who cared about compliance, and to organize themselves to react accordingly. The scores fitted well into the project because functionality and readability were both in the focus now, and it served the educational purposes as it supported learning.

Acceptable: Except for its potential of causing competition within the team and penalizing of functionality-focused behavior, the gamification was received well. Developers pointed out its positive motivating effects, and accepted it due to its fairness, quality of measurement, its "telling the truth" and the possibility to win as a team.

Influence: Developers who felt they were writing clean code right from the start felt they were influenced very little. Those who felt influenced named reasons like the desire to write clean code, wanting to win the prize or found what they learned about style helpful. Some developers changed their behavior to take care of style right from the start and to develop in a more organized way. B8 admitted to have actively influenced others so that everybody would have a high score. While Team A had to dedicate the last days before the deadline to fixing styles, Team B adapted their way of working early and did not feel influenced.

Additional effort: Developers who integrated automated checks into their normal way of working reported little additional effort. They used IDE plug-ins to do most of the work and accepted that they had to dedicate a little extra time. Developers who fixed problems later reported higher efforts. They spent whole days fixing their own and others' code. The finickiness of Checkstyle made them review their code several times before committing it to the repository. Using IDE functions to format code and Checkstyle themselves would have reduced this effort. Fixing compliance violations in code that they did not understand anymore was a major problem for them.

Readability improved: A controversy if readability had improved revolved around Checkstyle. Some developers accused it of not addressing issues like identifier naming but checking seemingly unimportant things like formatting. Yet, others found that quality improved indeed. Convention compliance made the code easier to read, while the required Javadoc comments added to understandability.

Difference: Broad consensus was that individual scores made a difference compared to one overall score. Reputation was first perceived as interfering with collaboration. But

	Skill	present	fair	important	understandable	likable	fitting	acceptable	influence	additional effort (inv.)	readability improvement	difference	concern (inverted)	overall average feeling
A1	2.6	0	- -	++	+	- -	- -	-	+	+	-	++	++	0 (2.9)
A2	—	+	-	0	+	0	-	0	-	- -	- -	+	++	0 (2.8)
A3	4.9	++	- -	+	-	- -	-	0	+	0	+	++	++	0 (3.0)
A4	—	+	++	+	- -	0	0	+	+	0	0	+	-	0 (3.3)
A5	—	0	++	++	- -	0	++	++	+	-	++	0	0	+ (3.5)
A6	3.9	+	0	-	+	0	+	-	-	-	-	++	++	0 (3.2)
A7	2.6	0	0	+	+	0	-	0	-	+	0	0	+	0 (3.0)
A8	3.9	0	++	++	++	+	++	++	-	- -	++	+	0	+ (4.0)
A9	3.4	+	0	+	0	0	0	0	0	-	0	+	+	0 (3.3)
B1	2.9	++	+	+	+	+	+	++	+	-	+	-	++	+ (3.9)
B2	3.8	++	++	+	+	+	++	++	+	-	+	++	0	+ (4.2)
B3	3.4	+	+	+	++	++	+	++	+		+		0	+ (4.0)
B4	3.5	+	0	+	+	0	+	+	+	+	++	+	0	+ (4.1)
B5	1.7	0	0	+	+	+	0	+	0	-	+	++	++	+ (3.6)
B6	3.4	+		+	+	+	+	+	-	+	+	+	0	+ (3.5)
B7	4.4	0	0	-	+	0	+	+	-	-	+	+	-	0 (3.2)
B8	4.8	++	++	++	+	++	++	++	++	0	++	++	++	++ (4.8)
Average A	3.6	3.67	3.11	4.00	3.11	2.67	3.00	3.33	3.00	2.44	3.11	4.11	4.00	0 (3.2)
Average B	3.5	4.13	4.14	3.88	4.13	4.00	4.13	4.50	3.50	2.71	4.25	4.14	3.63	+ (3.9)
Average	3.5	3.88	3.56	3.94	3.59	3.29	3.53	3.88	3.24	2.57	3.65	4.13	3.82	+ (3.6)

Figure 2: Likert-scaled opinions about the intervention, plus self-assessed skill (see Section 4.2)

developers reported to then have learned how to use it as a tool to see who needed help and to identify developers from whom they could learn. It inspired them to work hard to get a good reputation score, and A2 said that the "scores aroused some unexplainable competitive spirit" in him.

Concerns: Publishing scores was not seen as a privacy issue as long as scores were not published to others outside the team. Instead, developers explicitly stated that they had nothing to hide and even said that it helped them to organize themselves because it helped them to better understand their team.

5.4 Research Questions Answers

This subsection interprets the results with respect to the two research questions.

5.4.1 *Does Gamification of Coding Convention Enforcement Lead to Improved Compliance?*

Reputation depends directly on measured convention compliance. The history of the reputation scores clearly shows that the intervention had the intended effect on both development teams. It can be seen that for both teams, the individual and average reputation scores were lower at intervention start than at its end. The employed convention checking tool found (almost) zero problems at the end of the intervention for both teams as compared to their start. Furthermore, compliance increases for teams receiving the intervention (at End A, Team A better scores than Team B; at End B, Team B better scores than Team A). We therefore conclude that gamification of coding convention compliance leads to improved compliance.

The surveys additionally support this finding: In both teams, the intervention was perceived as present and highly important. Moreover, Team B and also Team A mentioned that they understood quite well what to do. They felt that the gamification had an active influence on them that would not have been there, if individual scores would not have been not computed. They noticed that it caused them additional effort. The developers found that the quality of their code improved due the intervention.

5.4.2 *Does Gamification Cause Undesirable Social Effects?*

The second research question whether gamification does cause undesirable social effects is more difficult to substantiate. The reason is that only indirect data from surveys and observations is available.

Indication that our gamification caused undesirable effects come mainly from Team A: the anecdote of the developer who got no sleep for one night, the breaking of previously working functionality before product presentation, and a slight overall tendency to dislike the gamification. Only A8 liked it, while A1 and A3 strongly disliked it (yet one of them, A3, later changed his mind when he recognized the benefits it brought to their code). Potential problems reported by both teams were that it caused some competition inside the team, was a bit unpredictable due to understanding problems, and made developers feel indebted to colleagues when those fixed their violations.

In Team B, the overall reception was very positive. For both teams the gamification was (totally) acceptable and welcomed. The only issues raised with acceptability (potential for competition and penalizing of mass contributors), were outweighed by motivating effects and perceived fairness. Only two developers were slightly concerned about individual scores, while all others were not. No developer reported a too strong influence on his behavior, thereby precluding the existence of extreme adverse effects like team friction or crowding out primary project goals. Quite to the contrary, developers helped each other to reach the goals. Developers did not report privacy concerns regarding the publishing of scores. The row averages indicate that no developer had strong overall adverse feelings, which would certainly have been the case, if something bad happened.

The results for Team A are not as good. This may have several reasons. The project managers (who had seen no reputation scores) gauged that, in general, Team A was weaker and less organized than Team B. Developers of Team A were mostly functionality-oriented. Moreover, they used a development framework that recommended slightly different coding styles, which led to controversial opinions about the usefulness of Checkstyle's conventions. By addressing their scores only lately, the hastiness caused additional troubles (broken features).

In summary, the gamification was perceived as rather fair, likable, fitting into the situation and acceptable, and causing only few concerns. Serious friction did not occur in the experiment; the broken features issue was probably due to a general organizational weakness in Team A. Additionally, as a lesson from this issue, Team B was more carefully introduced to the intervention, emphasizing that enough time was left. We therefore negate the question and conclude that gamification does not cause undesirable social effects.

5.5 Lessons Learned

Compared to Team B, Team A reported costs we deem unnecessarily high, and from which we derive our first lessons: One reason might be organizational weaknesses also observed by instructors that might have become intensified. Use caution when introducing gamification! Secondly, coding rules were not calibrated to the third-party libraries used. The enforced conventions should be adapted to the project situation! Thirdly, Team A tried to achieve perfect 10 scores to not risk missing the goal (which was intentionally not set to perfect scores). However, according to the eighty-twenty rule, the last few percent are the most costly ones to achieve. The goal was therefore intentionally set below the perfect score (to 9); but this must also be recognized and realized by the team! Fourthly, higher effort resulted from not using automated tools for fixing problems like formatting. Hence, available tools should employed by the teams where possible! Fifthly, Team A addressed violations not promptly but rather late, which is (i) usually more costly in itself, and (ii) leads to being able to benefit from readability improvements only lately. Weeks after answering our survey, A3 said he actually learned the value of cleaner code only after seeing its decay. So, gamification should reward and promote immediate adaptation of behaviors! Sixthly, for Team B, a positive difference between additional effort and benefits from readability improvements stands out. Feedback indicates that they had a relaxed and planned approach to dealing with reported violations. They fixed issues during programming, saw it as a normal task they had to dedicate time to, and used tools to automatically fix most problems. Gamification works best in a relaxed, planned but continuous way!

Some lasting learning effects may have occurred. The decay in reputation after the intervention for Team A supports that it was the intervention, which was responsible for the increase. But reputation does not fall abruptly after the intervention although much more code was produced. It only decreases by about 15%. This observation is congruent with the developers' statements that they had learned things like immediately writing comments, or keeping an eye on style.

The compliance report listed for each file the main contributor, the file's compliance rating and detected violations. This information helped developers to sufficiently well understand how scores were computed. The developers uttered concerns about the finickiness of Checkstyle. Still they found that it led to more readable code.

5.6 Summary

The CollabReview concept was successfully validated in two projects, where it fitted well into Scrum development. The primary goal of increasing coding convention compliance was clearly reached. It was seen as a good support for achieving better readability. Developers reported that they it made consider both functionality and readability when programming. The interventions were perceived well by the developers and we found only few indication of adverse effects. As a result of the experiment, a reasonable readability improvement was reached cost-effectively.

6. THREATS TO VALIDITY

Regarding internal validity, the evaluation was designed towards the gold standard of experimentation. Results obtained with and without intervention were clearly distinguishable. A between-groups and a within-group experimental design were realized. The developers had similar educational backgrounds were assigned randomly to the two groups in such a way that a similar average skill-level was present in both groups. Of course, both groups did not consist of perfectly equal individuals. The work context of both teams was comparable because the projects were similar with respect to process model (Scrum), tool support, programming language (Java), expected quality level (prototype), domain (web application), same project managers and the like. The size of the groups was quite small in statistical terms but a typical size for agile development. The results from the conducted surveys are congruent with statistical findings so that they support confidence. Developer opinions are evaluated partly based on averages of Likert data. There is an ongoing controversy about Likert data and parametric statistics. However, Norman concludes that statistics are sufficiently robust, save to use with Likert data, and that this assumption is backed by empirical literature [37].

Regarding external validity, the environment of the experiment was designed to reflect a typical development environment. But recruiting industrial development teams for experimental research is almost impossible [54]. The results of our research might therefore not simply extend onto highly professionalized industrial projects. Replication is necessary. However, besides such environments, there is a lot more software development going on in different domains that is often overlooked. For example, in teaching, start-up projects, amateur open source, or scientific programming (cf. [34]). Such environments are much more similar to the environment in which our experiment took place.

7. RELATED WORK

Coding conventions are often relied on as a first measure to improve internal quality (cf. [36]). Therefore, it is no surprise that they are a dime a dozen (cf. [31]). Famous examples are the GNU coding conventions or MISRA-C, which forbids constructs that tend to be unsafe. This is not to say that internal quality is all about coding conventions. For example, see the discussion of technical debt and its many derivatives like design, testing or documentation debt [13]. Regarding source code, researchers have looked into aspects like the problem of complexity (cf. [53]) and complementary

readability [14]. Nevertheless, an important ingredient to readability is coding conventions (see Section 2).

As of today, it is not possible to fully automatically assess the readability or quality of code. However, style violations and smells are significant hindrances for readability [50]. Consequently, readability can be improved through coding conventions that judged through style violations and code smells like white space, line length, comments, magic numbers and others. Buse and Weimer [14] showed that text-level analysis, on average, is better at judging readability than any human assessor. Lee et al. [31] analyzed several open source projects with respect to Checkstyle and the above readability metric. They empirically substantiated claims that consistent style and readability are correlated; i.e. that less readable source files have a higher rule violation density. Smit et al. [47] assessed the importance of 71 rules from code conventions for maintainability, and found that they can be discriminated into more and less severe rules. The metric we use for compliance is that of Smit et al. but without considering severity levels. Due to this important role of coding conventions, professional tools like SonarQube [4] or SQuORE [5] rely on conventions checking tools (e.g., Checkstyle) to assess internal quality. They even go as far as quantifying technical debt in dollars based on this. However, Boogerd and Moonen find that some coding rules can also have detrimental effects to reliability [12].

Various kinds of review processes, more or less formal ones, and distributed and asynchronous ones, using automatic scheduling, exist to enforce coding standards and guidelines (cf. [10]) Our approach to enforcing conventions differs from others in that it does not aim to enforce total compliance to conventions in each and every situation but still pushes developers towards compliance. For instance, Vashishtha and Gupta [52] and Merson [35] use Subversion hooks to prevent check-ins of code that is not compliant to conventions. The latter reports that introduction of new rules is problematic as it may stall the entire project until the new rule has been fully realized in the code. Plösch et al. combine static analyzers with downstream human reviews to improve code quality [39]. As opposed to this, Murphy-Hill et al. [20] propose soft advice based on a non-distracting ambient/background display to deal with the fallibility of automatic readability judgments.

Online communities like Coderwall or TopCoder [2,6] have long applied gamification to developers. In communities of web developers, increasing rewards leads to increased contributions of the desired kinds [19]. Anonymous bidding for implementations can even establish a market equilibrium for quality improvements [8]. Singer and colleagues apply gamification in development teams to promote good development practices like frequent commits [45,46]. Dencheva et al. used gamification to improve contribution to a knowledge management platform, taking quantity into account as well [17]. Contributor analysis as input to reputation systems is, for instance, researched in the domain of wikis [7, 27].

8. CONCLUSION

Source code is a medium of communication and the only precise documentation of implementation details in software development. It has therefore a central role for the preservation of valuable knowledge. Following coding conventions can make code more easily understandable for other developers by constraining expressive freedom and avoiding distracting code noise. However, this information exchange aspect is often neglected in practice. Investing efforts in making code more easily understandable has a high cost for the individual but benefits are spread over the whole team. This imbalance is a major reason for the appearance of the information exchange dilemma. We

- relate problems of unclean code to the information exchange dilemma,
- propose gamification as a way to motivate developers to comply to coding conventions,
- design a solution for implementing such gamification,
- explain how to integrate it into the social environment of a development process, and
- conducted two experiments with agile teams where gamification improved convention compliance, without major detrimental social effects.

Gamification was accepted as it gave developers a tool for self-management. Developers noted additional effort and potential hassles, which could, however, be controlled by better (self-)organization, support of respective tools, prompt fixing of violations, and not taking it too serious. Checkstyle was perceived as valuable although it has some shortcomings: being picky but dumb (e.g., not checking the sensibleness of identifier naming) and requiring adaptation to the peculiarities of the project. Overall, developers welcomed gamification for improving readability, for supporting self-organization, and for causing a lower additional cost than the value of its resulting benefits.

Our implementation may be better suitable for environments where rigid quality controls are not in place and where functionality deadlines are not too tight, e.g. academic teaching, amateur open source or scientific programming. At least, we are convinced that our study is representative for such environments. In the future, we want to replicate our experiments in other environments to make a transition towards industrial settings, and run trials with more long-term projects. Our results will hopefully serve us to convince further (industrial) development teams to try this approach.

Gamification may grant more flexibility for managing quality. It allows developers to fix each others' code, while effectively motivating them to invest in compliance themselves. It might replace rigid enforcement mechanisms for coding conventions in some cases. By motivating developers to invest in compliance, they have the flexibility to adapt their compliance efforts to other project pressures. This might be more efficient because if a fixed compliance level is set at a too low value, then it will not promote sufficient investments in compliance; if it is set too high, it will unnecessarily burden developers. Compliance efforts should be fair and the general conditions should be set in such way that the required efforts are borne by all and are not only at the expense of responsible team members.

9. ACKNOWLEDGMENTS

We like to thank the experiment participants for their cooperation, Andreas Zimmermann and Alexander Schneider for kindly hosting the experiments, Markus Eisenhauer for his advice with the experimental design, and Katrin Wolter and many anonymous reviewers for their helpful comments.

10. REFERENCES

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

[2] Coderwall. http://coderwall.com/.

[3] Collabreview. http://sourceforge.net/projects/collabreview/.

[4] Sonarqube. http://www.sonarqube.org/.

[5] Squore. http://www.squoring.com/.

[6] Topcoder. http://www.topcoder.com/.

[7] B. T. Adler, K. Chatterjee, L. de Alfaro, M. Faella, I. Pye, and V. Raman. Assigning trust to wikipedia content. In *Intl. Symp. on Wikis*, 2008.

[8] D. F. Bacon, Y. Chen, D. Parkes, and M. Rao. A market-based approach to software evolution. In *OOPSLA companion*. ACM, 2009.

[9] S. C. Bailin. Software development as knowledge creation. *IntJ. of Applied Software Technology*, 3(1):75–89, 1997.

[10] M. Bernhart, S. Reiterer, K. Matt, A. Mauczka, and T. Grechenig. A task-based code review process and tool to comply with the do-278/ed-109 standard for air traffic managment software development: An industrial case study. In *IntSymp. on High-Assurance Systems Engineering (HASE)*, pages 182–187, 2011.

[11] B. Boehm. Get ready for agile methods, with care. *IEEE Computer*, 35:64–69, 2002.

[12] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *ICSM*, pages 277–286. IEEE, 2008.

[13] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Future of Software Engineering Research*. ACM, 2010.

[14] R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

[15] C. Connell. Creating poetry in software code. *Boston Globe*, 1999.

[16] U. Cress, J. Kimmerle, and F. W. Hesse. Information exchange with shared databases as a social dilemma - the effect of metaknowledge, bonus systems and costs. *Comm. Research*, 33(5):370–390, 2006.

[17] S. Dencheva, C. R. Prause, and W. Prinz. Dynamic self-moderation in a corporate wiki to improve participation and contribution quality. In *ECSCW*. Springer, 2011.

[18] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: Defining "gamification". In *Intl. Academic MindTrek Conf.: Envisioning Future Media Environments*. ACM, 2011.

[19] D. DiPalantino and M. Vojnovic. Crowdsourcing and all-pay auctions. In 10^{th} *Conference on Electronic Commerce*. ACM, 2009.

[20] T. B. Emerson Murphy-Hill and A. P. Black. Interactive ambient visualizations for soft advice. *Information Visualization*, 12(2):107–132.

[21] F. R. Farmer and B. Glass. *Building Web Reputation Systems*. O'Reilly, March 2010.

[22] N. H. Frija. *The Emotions (Studies in Emotion and Social Interaction)*. Cambridge University Press, 1987.

[23] P. Graham. *Hackers & Painters: Big Ideas from the Computer Age: Essays on the Art of Programming*. O'Reilly & Associates, Inc., 2004.

[24] M. Harman. Why source code analysis and manipulation will always be important. In *SCAM*. IEEE Computer Society, 2010.

[25] R. L. Heneman and C. V. Hippel. Balancing group and individual rewards: Rewarding individual contributions to the team. *Compensation & Benefits Review*, 27(63):63–68, 1995.

[26] J. D. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2):16–20, 2001.

[27] B. Hoisl. Motivate online community contributions using social rewarding techniques — a focus on wiki systems. Master's thesis, Vienna University, 2007.

[28] W. S. Humphrey. *Managing Technical People*. Addison-Wesley, 1997.

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

[30] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[31] T. Lee, J. B. Lee, and H. P. In. A study of different coding styles affecting code readability. *IJSEIA*, 7(5):413–422, 2013.

[32] D. Leonard-Barton. Implementing structured software methodologies: A case of innovation in process technology. *Interfaces*, 17(3):6–17, May/June 1987.

[33] A. Lynex and P. Layzell. Organisational considerations for software reuse. *Annals of Software Engineering*, 5:105–124, 1998. 10.1023/A:1018928608749.

[34] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[35] P. Merson. Ultimate architecture enforcement: Custom checks enforced at code-commit time. In *Conference on Systems, programming, & applications: software for humanity*, pages 153–160. ACM, 2013.

[36] M. E. Nordberg, III. Managing code ownership. *Software, IEEE*, 20(2):26–33, March/April 2003.

[37] G. Norman. Likert scales, levels of measurement and the "laws" of statistics. *Adv in Health Sci Educ*, 15:625–632, 2010.

[38] C. Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *PPIG*, PPIG, 2010.

[39] R. Plösch, H. Gruber, C. Körner, and M. Saft. A method for continuous code quality management using static analysis. QUATIC, pages 370–375. IEEE Computer Society, 2010.

[40] C. R. Prause. Maintaining fine-grained code metadata regardless of moving, copying and merging. In *SCAM*. IEEE CS, 2009.

[41] C. R. Prause and Z. Durdik. Architectural design and documentation: Waste in agile development? In *ICSSP*. IEEE, 2012.

[42] C. R. Prause, J. Nonnen, and M. Vinkovits. A field experiment on gamification of code quality in agile development. In *PPIG*, 2012.

[43] P. Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009.

[44] B. Selic. Agile documentation, anyone? *IEEE*

Software, 26(6):11–12, Nov/Dec 2009.

[45] L. Singer and K. Schneider. Influencing the adoption of software engineering methods using social software. In *ICSE*. IEEE, 2012.

[46] L. Singer and K. Schneider. It was a bit of a race: Gamification of version control. In *Intl. W. on Games and Software Engineering*, 2012.

[47] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia. Code convention adherence in evolving software. In *Intl. Conf. on Software Maintenance*, ICSM, pages 504–507. IEEE, 2011.

[48] D. Spinellis. Reading, writing, and code. *ACM Queue*, October 2003.

[49] D. Spinellis. Code documentation. *IEEE Software*, 27:18–19, 2010.

[50] D. Spinellis. elyts edoc. *IEEE Software*, 28:104–103, March 2011.

[51] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering*, 2:201–207, 1997.

[52] S. Vashishtha and A. Gupta. Automated code reviews with checkstyle, part 2, 11 2008.

[53] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Software Eng.*, 14(9):1357–1365, 1988.

[54] E. J. Weyuker. Empirical software engineering research - the good, the bad, the ugly. In *Symp. on Emp. Softw, Eng. & Measrmnt*. IEEE, 2011.

[55] J. R. Whitson. Gaming the quantified self. *Surveillance & Society*, 1:163–176, 2011.

[56] S. Wray. How pair programming really works. *IEEE Software*, 27:50–55, 2009.