Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09719-4

Fostering good coding practices through individual feedback and gamification: an industrial case study

Matthieu Foucault[1] · Xavier Blanc[2] · Jean-Rémy Falleri[2] · Margaret-Anne Storey[1]

Published online: 20 May 2019

Abstract

Code quality is a constant challenge faced by today's software industry. To ensure that developers follow good coding practices, a variety of program analysis and test coverage tools are routinely deployed. However, these tools often fail to engage and change the practices of developers when applied to legacy systems as they output a huge number of warnings, quickly overwhelming the developers. In this article, we explore how individual feedback and gamification can motivate developers to pay more attention to good coding practices. To that extent, we implemented these two concepts in a tool that we deployed at two large companies where we conducted a case study. We find out that individual feedback is essential for motivating developers. We also find that gamification can be useful but has to be used with caution as it can frustrate some developers. Finally, we reflect on some lessons learned during our case studies, and conclude that the promising approach of our tool needs to be supported by longitudinal studies as well as comparative studies.

Keywords Industrial study · Software maintenance · Qualitative research · Developers motivation

Communicated by: Martin Robillard

✉ Matthieu Foucault
mfoucault@uvic.ca

Xavier Blanc
xblanc@labri.fr

Jean-Rémy Falleri
falleri@labri.fr

Margaret-Anne Storey
mstorey@uvic.ca

1 University of Victoria, Victoria, Canada

2 University of Bordeaux, LaBRI, UMR 5800, F-33400, Talence, France

1 Introduction

In today's software industry, many organizations have to maintain legacy software systems, which are "large software systems that we don't know how to cope with but that are vital to our organization" (Bennett 1995). One approach managers use to tackle this issue is to monitor the source code quality using two classical quality metrics: the number of *code smells* (Beck et al. 1999) and the level of *test coverage* achieved (Tikir and Hollingsworth 2002).

These quality metrics may be computed using dedicated program analysis tools, such as SonarQube[1] for code smells and Jacoco[2] for test coverage. These tools tend to be used at regular intervals or after each commit and report on any issues found, either through a browser or directly in the user's integrated development environment (IDE). These reports indicate where code smell issues are or which lines of code are not adequately covered by existing tests.

On top of these quality metrics, managers may also establish good coding practices for developers to follow, including *avoiding any new code smell*, *fixing critical code smells*, *fully covering new code fragments* or *aiming for at least 85% code coverage* (Williams et al. 2001). However, motivating developers to follow such good practices is difficult.

One reason for this situation is probably because reports produced by the program analysis tools contain too much information, most of which is unrelated to the changes authored by developers. For instance, when a developer opens a source code file to perform changes, the program analysis tools may pinpoint all the bad smells contained in the file (not only the ones authored by the developer) and all the uncovered lines as well, obfuscating the code and frustrating the developer. As a consequence analysis tools are unfortunately quite often ignored by developers (Johnson et al. 2013; Christakis and Bird 2016).

The other reason of this situation is certainly because software quality is often perceived as a fifth wheel. Developers are mainly motivated by developing new features but not by cleaning the source code or making better tests. Improving quality is a *de facto* not fun activity, which is reinforced by the attitude of managers who keep on imposing quality gates as objectives[3]

As a result, developers tend not to respect the good coding practices and even turn off the program analysis tools (see the Section 2, that lists the research studies that highlight this situation). They therefore may continue to commit less-than-ideal code that eventually leads to a decaying code base.

Two large French companies approached us in 2016 as they wanted to encourage their developers of legacy software systems to better use program analysis tools and follow good coding practices that would improve code quality. They asked us to suggest a dedicated solution to this motivation challenge with a special focus on addressing code smells and low test coverage. In this paper, we describe our case study with these two companies and present how we developed a solution that addresses poor developer motivation in terms of maintaining and improving code quality of legacy software systems.

Our collaborative approach to this research involved a design study (Sedlmair et al. 2012) methodology where we iteratively characterized the problem of poor developer engagement

[1] http://www.sonarqube.org/

[2] https://github.com/jacoco/jacoco

[3] A quality gate is defined by a threshold that drives the quality metrics. For instance, 85% of code coverage. When set, developers have no choice other than respecting the threshold.

in good coding practices for improving code quality, iteratively designed and implemented a solution to this challenge with ongoing input from users, and validated the solution using field studies.

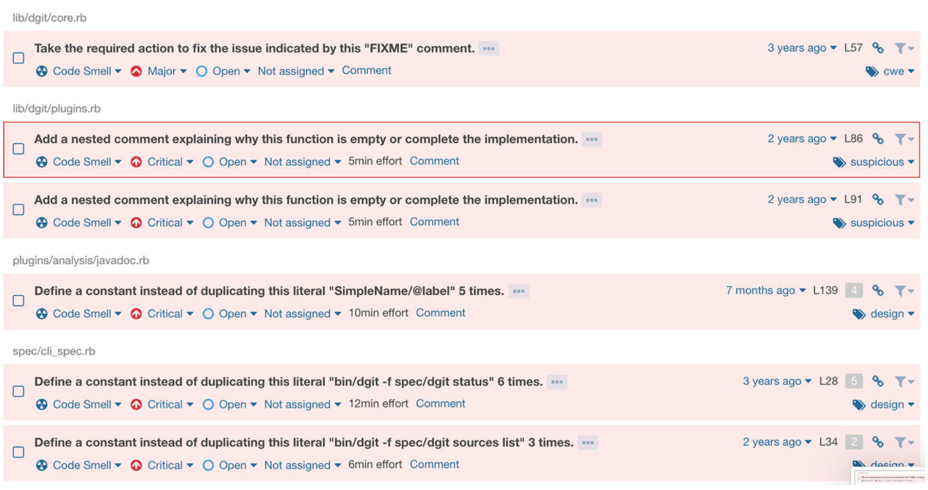
The solution we arrived at builds on two well-known key concepts: *individual feedback* (Nadler 1979) and *gamification* (Deterding et al. 2011). It is deployed in a tool called *Themis* that is plugged to program analysis tools and that provides individual feedback to the developers and integrates a gamification layer. Section 2 explains these two concepts and shows that, to the best of our knowledge, there is no large case study measuring their impact on developers' motivation to follow good coding practices in an industrial context.

In this paper we present such an industrial case study, and more precisely describe the evaluation of our solution with our two industrial partners. We sought input on the impact that *individual feedback* and *gamification* have on developers behavior through interviews and a questionnaire. Through this paper, we also share the two major impediments we faced while deploying our solution and share advice for other practitioners wishing to use similar concepts on how they may address these issues.

Our case study shows that:

- Individual feedback is perceived positively by all developers we interviewed. Its positive effects on code quality include an improved self-evaluation, an increase of awareness, positively changing coding habits and fostering team communication. To the best of our knowledge, no prior research work measured that effect in an industrial context.
- Gamification has a positive effect on some developers. It can increase fun and therefore the engagement in code quality tasks. However it should be used carefully: some developers are reluctant to play a game while working. This result confirms the results obtained by the previous studies. However, we also noted that some developers can take more interest in the game than in the underlying project and develop deviant behaviors. This is a new result that emphasizes the fact that gamification has to be used carefully in an industrial context.
- Managers may use both individual feedback and gamification to enhance coding practices, but two important aspects have to be enforced for them to work. Firstly, it is crucial to be very clear about how the individual data produced by the tool will be used. This confirms the result obtained by the previous study of gamification in an industrial context. Additionally, we observed that it is also very important to facilitate the use of the tool by creating new challenges, resetting scores, etc. This new result may help practitioners who want to deploy tools including a gamification layer in an industrial context.

The remainder of our paper is structured as follows: In Section 2, we present the related research. In Section 3, we describe the design of our study by presenting the industrial context behind the work, our research goals, and details of the research methods we used. In Section 4, we discuss the concept of individual feedback and how this was implemented in our solution. We also share how feedback following developer commits impacted their motivation and behavior in terms of using good coding practices. In Section 5, we present how gamification was implemented in our solution and how it influenced developers' behavior regarding good coding practices. In Section 6, we discuss the two major impediments we came across over the course of our case studies. In Section 7, we describe the limitations of our study. Finally, we conclude the paper in Section 8.



lib/dgit/core.rb

Take the required action to fix the issue indicated by this "FIXME" comment. — 3 years ago L57
Code Smell Major Open Not assigned Comment — cwe

lib/dgit/plugins.rb

Add a nested comment explaining why this function is empty or complete the implementation. — 2 years ago L86
Code Smell Critical Open Not assigned 5min effort Comment — suspicious

Add a nested comment explaining why this function is empty or complete the implementation. — 2 years ago L91
Code Smell Critical Open Not assigned 5min effort Comment — suspicious

plugins/analysis/javadoc.rb

Define a constant instead of duplicating this literal "SimpleName/@label" 5 times. — 7 months ago L139 4
Code Smell Critical Open Not assigned 10min effort Comment — design

spec/cli_spec.rb

Define a constant instead of duplicating this literal "bin/dgit -f spec/dgit status" 6 times. — 3 years ago L28 5
Code Smell Critical Open Not assigned 12min effort Comment — design

Define a constant instead of duplicating this literal "bin/dgit -f spec/dgit sources list" 3 times. — 2 years ago L34 2
Code Smell Critical Open Not assigned 6min effort Comment

Fig. 1 Example of the issues view of SonarQube on a third-party project

2 Related Work

In this section, we start by briefly describing what program analysis tools are. We then present existing studies that explain how developers use them. We then describe the concepts of feedback and gamification, and present existing studies that show their advantages and limits.

2.1 Program Analysis Tools

To help check the quality of their code, developers commonly use two kinds of program analysis tools: the *linters* and the *test coverage tools*.

Linters automatically identify *code smells* (Beck et al. 1999) and pinpoint parts of the code that should be fixed (Curtis et al. 2012; Letouzey and Ilkiewicz 2012). Some examples of linters are FindBugs (Ayewah et al. 2008) for Java or ESLint[4] for JavaScript.

Test coverage tools allow developers to discover which statements of their program have been executed by unit tests (Miller and Maloney 1963) and which parts are untested. Although research evidence on the utility of test coverage is mixed (Inozemtseva and Holmes 2014), several studies have acknowledged the fact that higher test coverage is correlated with fewer errors in the system (Mockus et al. 2009; Andrews et al. 2006), a metric frequently used in industry (and by our industry partners).

Much of the information reported by the linters is irrelevant to the developers and potentially obscures the one that they may find more meaningful to the task they are performing (Johnson et al. 2013). Figure 1 illustrates this problem with a screenshot of SonarQube where some issues are presented. Such a dashboard lists the issues and sorts them depending on their location, their rule, their criticality, etc. This situation is furthermore more problematic with legacy software systems which contain many code smells and uncovered lines of code.

[4] http://eslint.org/

Although the use of linters is widespread across the software development community, they are scarcely integrated in development processes, and not systematically used by developers (Ayewah et al. 2008; Beller et al. 2016). Johnson et al. (2013) interviewed 20 developers to investigate the reasons why they don't use linters more often and found that the most important reasons are the large number of and poor presentation of warnings, the high rate of false positives, the lack of collaboration support in the tools, and their lack of customizability. Similar findings were discussed by Christakis and Bird (2016) for industrial software systems, and by Fjóla Tómasdóttir et al. (2017) with a specific focus on JavaScript.

To the best of our knowledge, there has yet to be an empirical study investigating how test coverage tools are used in practice in an industrial setting. There is, however, evidence that testing is simply an activity that developers do not perform. Beller et al. (2016) performed an empirical study to find out how developers handle testing in the IDE, using the recorded activity of 416 developers. They discovered that most developers do not practice testing and do not run tests. They also found that tests and code do not co-evolve gracefully, and that developers usually run only a specific test in the IDE instead of the whole test suite.

After having reviewed the aforementioned research papers about how developers handle code smells and tests in practice, we can conclude that developers consider identifying smells or test-related tasks as a burden and are reluctant to perform them. This body of research reinforces the fact that developers should to be motivated to better use program analysis tools. They should be engaged so that the usage of program analysis tools is naturally integrated in their development process.

2.2 Developers Motivation: Feedback and Gamification

Motivation of developers was extensively studied in the past decades (Hall et al. 2008; Beecham et al. 2008; França et al. 2011). The various studies on this topic have identified a list of motivators that work well for developers. The main motivator that has been identified is that developers should *identify with the task* at hand. However this is hard to achieve for tedious tasks, such as the ones needed to maintain legacy software systems. We then choose to leverage on individual feedback and gamification since *feedback*, *rewards and incentive* and *recognition* are well established in the list of the known motivators.

The importance of the feedback has been largely studied in the past (Nadler 1979), showing its benefits on the learning process (Azevedo and Bernard 1995; Schooler and Anderson 1990). Further, the feedback should be as minimal as possible and actionable (pinpointing the errors and explaining how to fix them) to provide a strong added value as discussed by Anderson et al. (1996). However, to the best of our knowledge there is no industrial case study targeting the use of feedback to engage developers in better applying good coding practices.

Gamification is defined as the use of game design elements, such as points, score or ranking, in non-game contexts (Deterding et al. 2011). Gamification is a young domain where few theoretical foundations are available (Seaborn and Fels 2015). The emerging theories focus on player motivation, behavior change, and engagement, with specific attention paid to the relationship between intrinsic motivation (aligned with the player's inner values) and extrinsic motivation (coming from external factors) (Deci et al. 1999). The objective of gamification is to increase intrinsic motivation (i.e., become a better programmer) based on extrinsic motivators (i.e., gaining points). However, care must be taken to ensure that extrinsic motivators do not lead to decreased intrinsic motivation (Deci et al. 1999).

Research on the use of gamification in software engineering is relatively recent. According to the systematic mapping of Pedreira et al. (2015), gamification has been used in various software engineering activities, with an emphasis on software development.

Many studies use students to explore the effect of gamified environments. Prause and Jarke (2015) and Arai et al. (2014) showed that gamification for avoiding code smells seems to motivate students in an educational setting. Arai et al. (2014) found that students using the gamified tool removed more code smells than those using a non-gamified linter. However, they noted than some students were reluctant to play the game and preferred to remove code smells without the pressure of the score. Rojas et al. (2017) investigated the use of gamification to improve mutation testing. They performed an experiment involving students and noted that the students using gamification enjoyed writing tests more than the ones without, and wrote stronger test suites and mutants. Singer and Schneider (2012) describe a system using points, badges, and leaderboards to provide an incentive for developers to commit their code more often. After conducting an experiment with 37 students, their interviews showed that the tool increased the participants' awareness of the other developers' activity, and was effective in making some participants make more frequent small commits.

To our knowledge, only two studies explore the use of gamification in an industrial context. In the experiment from Prause and Jarke (2015), developers reported that they were more attentive to the readability of their code when developing, thanks to the presence of a gamified tool. The tool was well perceived by the developers and the authors found only few indications of adverse effects. Snipes et al. (2014) explore the effect of gamifiying the IDE to improve the knowledge of advanced code navigation features among industrial developers. They find out that generally over 50% of developers are interested in using gamified tools, and that point score and leaderboard was the more effective gamified motivators.

Finally, Barik et al. (2016) present a discussion about how gamification is usually applied in software engineering activities and find out that it is always applied in a narrow setting, restricted to points, badges and leaderboards. Dal Sasso et al. (2017) then describe a "framework" to explain how to gamify software engineering activities.

Our study contributes to these fields (feedback and gamification) by providing a large industrial case study. In our study, we note that individual feedback and gamification raise awareness of the code smells and coverage issues. We also find out that gamification can motivate developers to change their code writing habits. It confirms most of the results observed by previous studies showing that gamification may provide benefits in engaging developers to better apply good coding practices. It also confirms that some developers are reluctant to use a gamified tool. Finally our study pinpoints two impediments that may be faced when deploying such an approach in an industrial context (the usage of individual data and the need for facilitation).

3 Study Design

The study we present in this paper followed an iterative process, as illustrated in Fig 2. We were initially contacted by Pôle Emploi early in 2016. At that time, our initial goal was to understand how we could motivate developers to follow good coding practices. Following initial discussion with Pôle Emploi we decided to explore how interventions concerning individual feedback and gamification could impact developer habits. We then designed and developed a tool, called *Themis*, that puts together these two concepts. A first version was deployed late 2016 at Pôle Emploi for a period of three months. Initially, individual feedback and gamification features were dependent on one another. At the end of this period, we

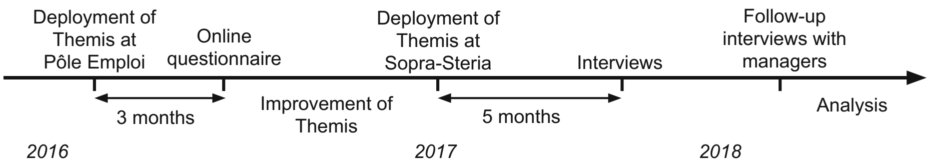

Deployment of Themis at Pôle Emploi
Online questionnaire
Deployment of Themis at Sopra-Steria
Interviews
Follow-up interviews with managers
3 months
Improvement of Themis
5 months
Analysis
2016
2017
2018

Fig. 2 Timeline of our case studies

surveyed the developers who used *Themis*: we aimed to understand how they used the tool's different features in terms of the feedback it provided, as well as its gamification elements. It should be noted that Pôle Emploi decided to stop the use of *Themis* at the end of the three months period because the developers felt like it was not mature enough (problems with the game design, bugs with the version control system connector, etc. see the Section 6).

Based on the results of our survey and discussions with our partners at Pôle Emploi, we changed our design. The second version separates the individual feedback and gamification features. This is the design we present in this paper. In 2017, we deployed the new version of *Themis* with our second industrial partner, Sopra-Steria, where it was used for a period of five months. To understand the impact of this deployment and to gain further insights about how individual feedback and gamification can engage developers on code quality issues, we conducted face-to-face semi-structured interviews with the developers that used the tools. It should be noted that Sopra-Steria is still using *Themis*.

Finally, after a period of 6 months, we performed semi-structured interviews with one manager at Pôle Emploi and one manager at Sopra-Steria to debrief about the use of *Themis* in their projects.

This section then presents the context of our two industrial partners and explains the method we followed to get their insights.

3.1 Context of the Industrial Case Studies

Pôle Emploi and Sopra-Steria faced the challenge of developing a culture of software quality that would help motivating their developers to follow good coding practices. Even though they provide analysis tools for their developers (such as SonarQube for instance), they felt the need for other means to better motivate their developers regarding software quality.

3.1.1 Case Study 1: Pôle Emploi

Pôle Emploi is a French governmental agency that provides financial aid for unemployed people (5.5 million people as of February 2017). It has 50,000 employees among 900 offices in France and a website that receives over 45 million visits each month. The business depends on a software platform composed of several sub-systems, maintained daily by 300 developers. Our study focuses on a central component of the platform that has a major financial impact on their users. This central component was initially deployed in 2006 and consists of 550k lines of code (in 2016). It is a J2EE application that is maintained by a group that includes 1 manager, 2 team leads, and 14 developers (divided into a team of 11 developers and a team of 3 developers). The manager and the two team leads were seniors (more than 10 years of experience). The developers were all software engineers with 5 to 10 years of experience, and could be considered to be expert developers. The team shared a same software engineering culture: Object Oriented developers (Java), concerned about good design (patterns) and high code quality, connected to the end users (iterative process) and therefore willing to provide software system with no bug.

Back in 2006 when Pôle Emploi started to develop this central component, no linter was used nor test coverage tool. After several years, Pôle Emploi set-up company wide quality policies and tools (mainly SonarQube) to be applied only to new developments. However the manager responsible of this central component was very interested by code quality, and therefore wanted to use these quality policies and tools on it, leading him to deploy a custom version of the company quality policies and tools (dedicated to the component). Furthermore, he decided to give a five-days training on clean-code principles to all of the developers.

After several months, even using this customized version of SonarQube, there were still too many quality issues reported by the tool, mostly on naming convention rules that are very tedious to fix. A consensus was then reached between the manager and the developers to focus on the newly developed code with the main objective to not introduce any new issue. To enact this vision, a large display was installed in the development open-space showing the advancement of quality issues. The display showed a curve reflecting the number of quality issues in real time. The curve was increasing when new quality issues were created, therefore implicitly asking developers to fix them. Thanks to this initiative, the manager and team-leads discussed regularly with the team about code quality, especially when the display showed that new quality issues were added to the system.

This initiative succeeded in improving the engagement of developers about quality issues, and the manager wanted to explore new ways to go further on this topic and contacted us in the beginning of 2016.

3.1.2 Case Study 2: Sopra Steria Région SudOuest

Sopra Steria is a European information technology consulting company that develops large software systems for its customers. Sopra Steria has more than 40k employees among 40 countries. Our study focuses on a group composed of 50 developers located in Bordeaux, France. That group is lead by a senior manager and two senior team leads (more than 10 years of experience). They shared a same software engineering culture: concerned about good design (patterns) and high code quality. Furthermore, they were all highly driven by the client satisfaction. The developers were mostly junior engineer developers (from 1 to 5 years of experience). The software engineering culture among the developers was highly heterogeneous.

In 2016, that group was assigned to the maintenance of a huge legacy software component (several thousands man days) owned by a company whose name is confidential. The development of this component started in 2007 when the owner asked a third party company, whose name is also confidential, to handle the software project. That third party company developed and maintained the component until 2016, when the owner decided to move and asked Sopra Steria to handle the project. Regarding quality concerns, we were not able to know what was the policies and tools used by the third party company. We do know that the owner of the software component used SonarQube in 2016 and noticed thousands of rule violations. When the owner contacted Sopra-Steria, a contractual engagement was signed and stated that the number of violations could not increase further.

When they took over the development, Sopra-Steria applied all the company's best-practices about quality management, including the use of a SonarQube installation configured for the project as well as regular training on the code quality topic (some mandatory, some optional) for all developers. To ensure that the developers followed the quality rules of SonarQube, the team leads were performing regular feedback meetings where they were discussing with the developers about the violations found by SonarQube.

However, the manager and team leads found that preparing these meetings was a lot of work and therefore wanted to try tools that could save the time of manually reviewing the code and the quality reports to prepare them. Therefore, they contacted us in the beginning of 2017, and decided to deploy *Themis*.

3.2 Methods

As shown in the Fig. 2, we distributed two different questionnaires to members of Pôle Emploi: one version of the survey was sent to the manager and two team leads (referred to as managers in the rest of the paper), and another version of the survey was sent to the 14 developers. All three managers as well as 8 out of the 14 developers responded to the surveys we sent. All answers were provided anonymously (unless participants chose to give us their email address) and participants were not given any incentive to answer the survey—they generously spent time answering questions without compensation to help us understand the effects of the tool they were using.

The surveys included closed- and open-ended questions focusing on the different information views provided by *Themis*, asking how (and how often) developers and managers used the different features. Other questions were related to how important gamification is to developers, and whether they noticed a change in their coding practices since *Themis* was introduced. Through follow-up questions sent to willing participants (by email), we were able to learn that the insights we gained from our analysis of the survey responses resonated with the research participants and helped to confirm our findings from the survey. In anonymizing the responses, we were sensitive to maintaining the confidentiality of the individual developers. A copy of our survey questions, interview guide, and ethics approval are available online.[5]

Our analysis of the survey responses allowed us to improve both the individual feedback and gamification features of the tool. While initially combined, we also separated the two features as not all developers were enthusiastic about the gamification functionality.

This improved (second) version of *Themis* was deployed with our second industrial partner, Sopra-Steria. Our first partner having ended their trial period of *Themis*, we were unable to conduct a second study with them. Five months after *Themis* was deployed with all teams at Sopra-Steria, we were given the opportunity to conduct face-to-face interviews with developers and team leads. Two team leads and six developers volunteered to participate in semi-structured interviews, lasting no more than 25 minutes each.

For the analysis presented in the paper, we combined the responses from both industrial case studies, focusing on the open-ended questions. Some of these questions are shown in Table 1. Our goal is to precisely understand how personal feedback and gamification affect developers on the code quality topic, on a qualitative level. Even though the implementation of *Themis* differed slightly between the two case studies, the core principles and goals remained the same, and the findings from the first case study remained relevant for our purpose. To analyze our data, we resort to a card sorting methodology (Spencer 2009). Using the responses from both studies, we produced 280 cards, each containing a statement expressed by a participant that described how they used the tool, what they learned from it, how they felt about gamification, etc. When quantifying the statements, i.e. when using

[5] https://github.com/thechiselgroup/GamifyTechDebtData

Table 1 Examples of questions asked in the online survey or the interviews

What do you expect from using *Themis*?
Does *Themis* help you to prevent or fix technical debt?
Does *Themis* help you to learn?
Were you worried that *Themis* will grade the quality of your code?
What information do you find useful in the home dashboard?

terms such as "most developers", we referred to the number of participants mentioning a specific behavior or theme, using our cards to record the number of occurrences of each theme. Three researchers (three authors of this paper) worked iteratively to sort the cards into categories: these categories appear in the findings described in the remainder of this paper. The card sorting activity is illustrated in Fig. 3.

4 Providing Individual Feedback

In this section, we present the feedback model used in our case study and finally explain how it helps developers achieve clean code and improve test code coverage. (The gamification feature is separately discussed in the next section.)

4.1 The Feedback Model of Themis

In order to produce individual feedback, *Themis* performs an analysis of each commit with respect to adherence to the code coverage and clean code coding practices. Such an analysis is based on the reports provided by program analysis tools (such as Jacoco for code coverage and SonarQube for clean code). Themis links the analysis reported by these tools to the authors of the commits, and then gives them semantics. More precisely, every file changed

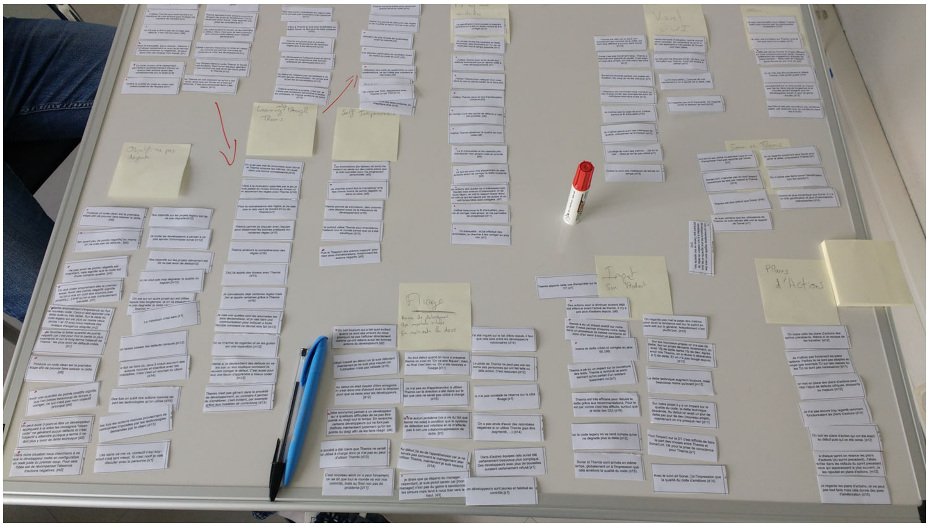


Fig. 3 Card sorting activity: 280 cards were laid out and two researchers classified and annotated the qualitative data over several iterations

Table 2 Definition of the *Themis* actions for clean code and code coverage practices

Coding practice	Healthy action	Harmful action	Repairing action
Clean code	Does not add nor removes code smells	Adds more code smells than it removes	Removes more code smells than it adds
Code coverage	The file's coverage was above a set threshold (by default 80%), and the action does not reduce the coverage	Decreases the coverage of the file	The file's coverage was below a set threshold, and the action increases it

An action describes a change on a single file and whether it follows a given coding practice

through the commit is tagged with a *clean code action*, and all non-test files are tagged with a *code coverage action*. These actions are then categorized as: *healthy*, *harmful*, or *repairing*. The meaning of these categories (when applied to the two types of code practice actions) are listed in Table 2.

For example, if a developer creates a commit where the files *Customer.js* and *TestCustomer.js* are modified, the following action tags will be created:

- A *clean code* action for *Customer.js*.
- A *clean code* action for *TestCustomer.js*.
- A *code coverage* action for *Customer.js*.[6]

Then, *Themis* categorizes these actions in the three aforementioned categories—*healthy*, *harmful*, or *repairing*—depending on the impact of the developer's change on the corresponding coding practice.

To further illustrate how *Themis* defines actions and categorizes them, the listings 1 and 2 present two versions (before and after, respectively) of two JavaScript files that have been changed through a single commit. This simple source code aims to create customers with first and last names. The second version adds a type check on one of the parameters. The green and red marks on the left of the listing represents whether the line is executed by a test or not, respectively (only line 4 in Listing 2 is not executed by a test). The underlined line of code (line 3 in Listing 1) has a code smell: a logging library should be used instead of *console.log*. Note that these adornments shown in the IDE are added by the code coverage and code smell analysis tools and are shown for all such issues in the code (not just for the code the developer commits).

By analysing this commit (from listing 1 to listing 2), *Themis* creates and categorizes the three following actions:

- A repairing *clean code* action is created for *Customer.js*—the *console.log* call-in line 3 was removed.
- A healthy *clean code* action is created for *TestCustomer.js*—the new line does not introduce or remove bad smells.
- A harmful *code coverage* action is created for *Customer.js*—the added line (line 4) is not covered by the test.

We mentioned in Section 3 that the design of *Themis* was improved between our two case studies. The concept of identifying developer actions remained the same, but its

[6]Themis does not create any *code coverage* action for test code, which explains why there is no such action in *TestCustomer.js*.

```
//Customer.js
🟩function createCustomer(firstName, lastName) {
🟩  console.log("a customer was saved");
🟩  return { firstName, lastName }
🟩}

//TestCustomer.js
function testCreateCustomer() {
    var customer = createCustomer("Bob", "Sponge");
    assert(customer.firstName).equals("Bob");
}
```

Listing 1 First version of two JavaScript files: one that contains a simple function to create a customer and another one that tests it

implementation was refined. The granularity of the actions was initially much finer: each instance of an added or removed code smell resulted in a *positive* or *negative* action, respectively. For example, removing a line of code that had three code smells was tagged with three positive actions. In the refined version of *Themis*, it is just tagged with a "repairing clean code action", which we found was easier for developers to follow. Also, the concept of healthy actions did not exist previously, so if developers wrote code that did not contain any code smells, this information did not appear in *Themis*. We found that showing healthy actions helps motivate developers to avoid adding code smells.

Themis was designed to be used mainly by developers. The main view shown to a developer when they open *Themis* contains a summary of their number of healthy, harmful, and repairing actions in each of the projects they contribute to, and an action feed listing their last actions, as illustrated in Fig. 4.

When they perform harmful actions, developers can click on the action to see the infringing code and, in the case of harmful and repairing *clean code* practices, they can view a detailed description of the rules they broke and see examples that show how to rewrite the code in a compliant format.

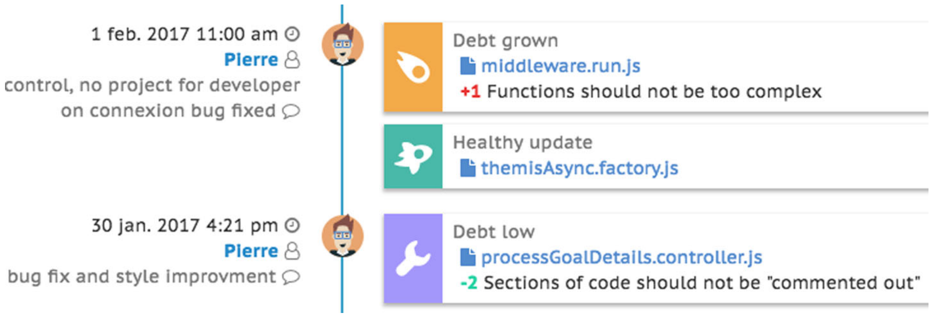
4.2 Advantages and Limits of the Themis Individual Feedback Model

To better engage developers in improving the software quality, we propose individual feedback to developers that may be perceived as more actionable. The core idea is that

```
//Customer.js
🟩function createCustomer(firstName, lastName) {
🟩  if (!_.isString(firstName)) {
🟥       throw "The customer needs a first name"
🟩  return {firstName, lastName }
🟩}

//TestCustomer.js
function testCreateCustomer() {
    var customer = createCustomer("Bob", "Sponge");
    assert(customer.firstName).equals("Bob");
    assert(customer.lastName).equals("Sponge");
}
```

Listing 2 Second version of the two JavaScript files. A type check was added in createCustomer, and the value of lastName is now checked in the test


1 feb. 2017 11:00 am
Pierre
control, no project for developer on connexion bug fixed

Debt grown
middleware.run.js
+1 Functions should not be too complex

Healthy update
themisAsync.factory.js

30 jan. 2017 4:21 pm
Pierre
bug fix and style improvment

Debt low
processGoalDetails.controller.js
-2 Sections of code should not be "commented out"

Fig. 4 Action feed for the developer Pierre. The first commit contain one harmful and one healthy action. The harmful action violates one rule (as indicated by the +1). The second commit contain one repairing action that removes two rule violations (as indicated by the -2)

information about the quality of the code changed in a particular developer's commits should be highlighted in their code quality report, as opposed to presenting information about the quality of all of the code, especially code the developer didn't touch.

In both case studies, *Themis* was introduced in the teams' development environments and the use of the tool was voluntary. Out of the 14 developers that used *Themis* and participated in our surveys or interviews, 12 reported looking at the *Themis* feed at least once a day, most of those either first thing in the morning or after lunch. Below we report the themes that appeared in our classification of the developers' answers to our survey and interviews.

4.2.1 Individual Feedback Brings Self Evaluation

Individual feedback is perceived by participants as a mechanism to perform a better self evaluation: *"I really see what I did myself, but on our linter we have everything that is raised'. It is more personal, therefore it's better."*[d_{12}], although this report of being "better" is highly subjective and not shared by all developers. Developers reported using the action feed to *"evaluate [their] work very quickly"* [d_6] and *"see [their] areas for improvement"* [d_1]. Overall, the developers felt that individual feedback allowed them to see whether they followed the good coding practices. They didn't have to filter through information that is reported by the analysis tools, i.e., code that they did not write—the developers could therefore identify how *they* followed the coding practices.

4.2.2 Individual Feedback Raises Awareness

Participants reported that individual feedback makes software quality more concrete, and therefore raises awareness: *"The individual feedback helped [them be] aware of the importance of some practices"*.[d_{13}] One participant even reported that he *"did not think about following the practices everyday* "[d_{12}], expressing the fact that individual feedback now reminds him to follow the practices. However, the message transmitted within the feedback should be as clear as possible for the developers to take advantage of them. One participant said that *"sometimes he can't find the errors that cause the harmful actions in the source code"*.[d_{15}] The feedback should therefore be meaningful otherwise it may quickly be considered as "noise".

4.2.3 Few Learning of Good Coding Practices

Most participants reported that they already knew the coding practices, and therefore did not acquire new knowledge. However, some participants did report that they learned some coding rules, particularly related to *code smells*. One participant explicitly mentioned the *"correction templates"*[d_{16}] as an important part of the tool when it comes to learning new rules. The participant also reported that he *"was not convinced with some rules."*[d_{16}]. An interesting tidbit is that the correction templates (small examples explaining how to fix the rule) as well as the rules are provided by the program analysis tools (such as SonarQube), and not by *Themis*. Themis only presents them within the individual feedback. These comments from d_{16} show however that the developers pay more attention to the rules thanks to the individual feedback.

4.2.4 More Caution to Code Quality in Coding Habits

We identified two factors that changed the behaviors of developers: the increased awareness and knowledge mentioned above, and the fact that the quality of one's work was attached to their actions (*"the harmfulness can only increase, not decrease"* [d_{12}]). A participant explained how the feedback from *Themis* lead them to follow good coding practices: *"It does not disrupt the development process, on the contrary. When we code we think about not creating any harmful actions, therefore we have less [negative] feedback, and we spend less time fixing mistakes."*[d_{12}] In both iterations of the study, developers reported paying *"specific attention every day before committing code"* [d_2] as well as using *"quality measurement tools available to [them] before committing [their] code."* [d_6] This effect is also seen with the monitoring of code coverage as one developer reported: *"We force ourselves to add unit tests, we are more careful."*[d_{13}]

4.2.5 Quality Is Promoted As A Teamwork Concern

One participant mentioned that *"I go to check which issues are triggered the most, and I send a message to the whole team to explain how it should be done."*[m_{12}] Some participants also use monitoring features to update and communicate with their co-workers: *"the actions reports at the end of the sprint [...] are useful to me in order to [...] provide reminders to the team or individuals, if needed."* [m_3] This shows that individual feedback is not only individual but also promote collaboration. One participant clearly express that point: *"It gives birth to team discussion with the goal to promote and disseminate good coding practices."*[d_{15}]

5 Gamification

In this section we first describe the gamification model we propose with the main objective to increase intrinsic motivation based on extrinsic motivators (Deterding et al. 2011). We then describe our findings relevant to the use of this gamification model in our case study.

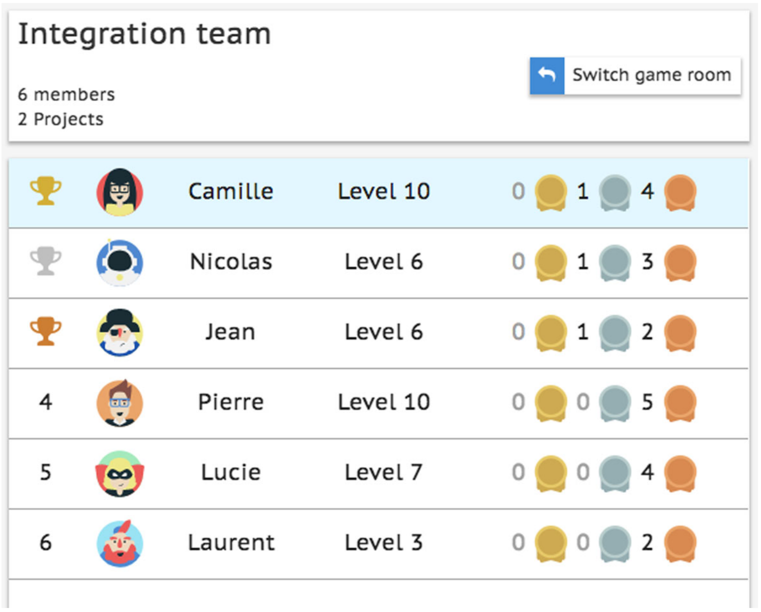
5.1 The Gamification Model of Themis

The gamification aspect is an opt-in feature: if a developer wants to use the gamification layer and compete, they can join (or create) a *game room*. A *game room* gathers a set of developers that participate in a game and agree to share their scores and be ranked in a shared leaderboard. This *game room* concept was designed after analyzing the feedback we received from the first case study conducted with Pôle Emploi, where we found that developers prefer to have the freedom to join a game (or not) and feel more confident when they know who can (and cannot) see their scores and rankings. Figure 5 presents the game room view.

Our gamification model provides developers with two different games. The first game consists of a way to earn points and achieve levels in the game— the more healthy and repairing actions a developer performs (in terms of code smells and test coverage), the more points they earn. By default, harmful actions do not lead to a developer losing points, but this option can be configured for a more difficult game if they wish. These points and levels are a measure of how well someone does with respect to good coding practices.

The second game supports developers in winning badges based on the type of actions a developer performs. There are five type of badges, each with three levels (bronze, silver, gold) and harder requirements for each level:

- **Steadfast:** the developer must perform a streak of actions that are either healthy or repairing.
- **Samaritan:** the developer must perform a specified number of repairing actions.
- **Meticulous:** the developer must perform less than a specified number of harmful actions.
- **Fair:** the developer must perform a given number of healthy actions.
- **Accomplished:** the developer must receive a combination of all the other badges to achieve this badge. Furthermore, to have a silver or gold accomplished badge, the developer must have the four other silver (or better) or gold badges, respectively.



Integration team

Switch game room

6 members
2 Projects

🏆		Camille	Level 10	0 ●	1 ●	4 ●
🏆		Nicolas	Level 6	0 ●	1 ●	3 ●
🏆		Jean	Level 6	0 ●	1 ●	2 ●
4		Pierre	Level 10	0 ●	0 ●	5 ●
5		Lucie	Level 7	0 ●	0 ●	4 ●
6		Laurent	Level 3	0 ●	0 ●	2 ●

Fig. 5 A game room where six developers are playing

The ranking in a room is shown thanks to a leaderboard. The ranking is computed by badges first (similar to the Olympic Games), and, in case of equality (same badges), the levels are used. Figure 6 presents Camille's rankings and badges: she is at level 10 with 450 points (next level is 660 points), has four bronze badges, and the silver Samaritan badge. She is first in the room because she has more badges than the other players. Pierre (4th in the ranking) is level 10 but has less badges than Camille, Nicolas and Jean.

To make the room more fun, the badges are periodically reset, which allows new players to join games and challenge the more experienced players.

As mentioned above, the gamification feature was not opt-in in the first version. The game was also much simpler: each positive action gave points to the developer, while each negative action they performed reduced their points. The developers in our study found this to be rather tedious and it led to an endless competition where the best scores could be obtained by performing mass corrections of code smells. Although the game was different between the two versions, much of the feedback we received was relevant to our second research question, and thus we report it below.

5.2 Gamification Findings

Here we present the findings of the survey and interviews that consider the gamification layer in *Themis*.

5.2.1 Gamification Affects Mood (Mostly Positively)

The team leads and developers from our study were instrumental in setting a positive culture for the use of gamification. In particular, the teams did not take "the game" too seriously and they enjoyed an atmosphere of *playfulness*, as one of the developers shared with us: *"[The game room] allows [us] to figure out who will bring chocolate croissants at the end of the sprint"*. [d_5] Rather than saying a certain developer was "last", they could joke that they owed the team chocolate croissants. Participants found the gamification fun without it being a clear motivator: *"the badges themselves are not a goal [that I have], but it's fun, especially to obtain those we never had "* [d_{11}] Across all the developers who agreed to answer our survey or participate in the interviews, only one person did not wish to use the *Themis* gamification feature: *"I don't really like the idea of a game room. [...] I think it is a pity that you need to have a reward to code properly."* [d_7] Such consideration shows that gamification *must* be an opt-in feature, and no one should be forced to participate in a game room.

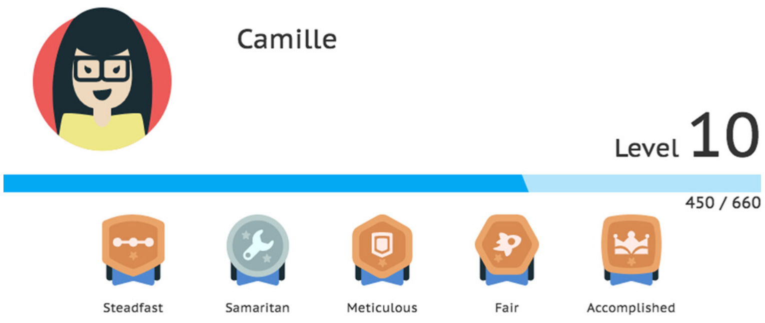


Fig. 6 Camille's level and badges

5.2.2 Leaderboard and Badges Induce Comparison

Some developers mentioned using the leaderboard to *"position [themselves] relatively to [their] colleagues, not with the goal to show that [they are] better than them, but to see if [they are] as good as them."* [d_6] The gamification then promotes good coding practices and induces personal or team challenges: *"I use levels and badges to challenge myself in the team"* [d_{15}]. One participant clearly expressed that communicating about healthy actions is gratifying: *"There is a better consideration of the healthy actions. The transparency of the game room calls for meticulousness."*[d_1] Some participants did not necessarily see the badges as a motivator, but looked at the game room to check whether they produced the same level of code quality as their teammates. One developer said that *"by following Themis' suggestions, the badges always end up coming"* [d_{16}].

5.2.3 Game Goals Impact Project Goals

We found that many developers were "competitive" by nature and that they felt the extrinsic reward of the game was a motivator to improve their code quality and perform corrective actions. These developers reported that the levels and badges were the first items they would look at when using *Themis*, and that they would make code changes that were clearly motivated by badges rewards. The main strategy used to win badges was to be opportunistic: *"We use our linter to fix the mistakes in the code and thus win points. We do this mainly in the classes we are working on."* [d_{14}] This strategy was mentioned by several developers who all pointed out that they limited themselves to performing repairing actions in the vicinity of the source code they were already modifying. A participant pointed to a possible adverse effect of gamification: *"some people spend time fixing classes that are historically not modified only to win a few points when they have pending development/debugging [tasks]."* [d_5] However, this negative feedback was mitigated by our discussions with the team leads, who assured us that this behavior was incidental at worst. Further, we didn't hear of any other strategies used to win points and the team leads we interviewed confirmed that they *"did not observe any unconventional behavior to win points or badges."* [m_{11}]

5.2.4 Game Design Matters

We already explained that the first version of *Themis* deployed at Pôle Emploi had a poor game design. In particular, one participant expressed the fact that: *"a large difference of levels may discourage developers to play, and even make them stopping their engagement in software quality."*[d_3] This is why we decided to provide a brand new game design with more games in the second version of *Themis*. Some bugs in the game design or in the tool may frustrate developers and even change their behavior. For instance, in the second version of *Themis*, a bug was affecting wrong actions in case of git merge. A developer complained about this bug: *"The merge bug made me lose some healthy actions. It prevented me to be first"*.[d_{12}] Another developer even changed his way of making commits: *"There is a bug with the git merge, so I try not to merge anymore"*[d_{14}] We therefore quickly fixed that bug to avoid more frustration. The current game design considers that the developers all perform quite the same amount of work. However one participant noted that *"The level and the badges are biased because they depend on your number of commits. However one may work part-time"*[d_{15}]

6 The Two Major Impediments

In this section, we present the two major impediments we came across to deploy our solution in an industrial context. The insights we present in this section were discussed during the follow-up interviews with the managers of Pôle Emploi and Sopra-Steria.

6.1 Developers Need a Clear Understanding of How the Data is Used

Managers may have access to a variety of information, such as the number of harmful actions performed by a team or an individual. Although this information is available in other tools, the way it is presented in *Themis*—showing "good" and "bad" actions—may make it easier for leads to evaluate a developer's "skill" or to judge the "quality" of their work. Evaluations that are based on such limited information can not only be inaccurate, but they can also pose a major ethical issue.

That said, we did not hear any concern with this potential misuse of the data in either case study we conducted. Before *Themis* was deployed in the first case study, the issue was briefly and informally raised with managers of the company, but no formal document was created to define a data usage policy. In the second case study, the company defined a data usage policy in a more formal manner, and developers were made aware of it during a presentation before *Themis* was introduced to them

However, in both case studies, developers reported that they were initially worried about being evaluated through *Themis*. This fear of being evaluated quickly disappeared when the developers understood that the goal of the tool was to help them, and that no performance assessment was going to be made. Nevertheless, this initial fear is an important issue that needs to be addressed if similar tools are to be deployed in other settings.

As the designers of *Themis*, we are also preparing a generic data usage policy to ensure that the conditions of data usage are well defined. This policy will be provided to companies wishing to use the tool in the future. We also intend to make the policy directly visible to users in *Themis*, such as when the tool starts up. This is particularly important as future users of *Themis* may not be available for a demonstration of the tool and presentation of these policies, or they may join the teams that use the tool after it is introduced.

6.2 An Organizer Should Facilitate the Gamification

Both Individual Feedback and Gamification aim to improve the engagement in Software quality. These two concepts rely on human aptitudes and therefore should be organized and facilitated.

Based on our experiment, we observed that one member of the team should play a specific role, we named *organizer*. The organizer has the responsibility to make all the developers react to the individual feedback they receive. At the beginning, the organizer should therefore explain to all the developers how to read the individual feedback and react to it. We further observed that some developers prefer to receive feedback after each commit, although others want to be notified once or twice a day at most. The organizer should therefore explain the developer how to configure their feeds. Regarding gamification, the organizer should create a positive atmosphere and involve the developers in game rooms. They should also put some rhythm into the game by resetting the score or adding some challenges.

It should be noted that this role of organizer appeared organically at Sopra-Steria. Two developers (technical leads) were so enthusiastic that they deliberately decided to organize the team. Thanks to the discussion with the manager of Sopra-Steria, we realized that this role was mandatory. Likewise, the discussion with the manager of Pôle Emploi brought up the lack of an organizer as an issue in their case. We therefore now ask for volunteers that want to play this role when we deploy *Themis* in a new industrial context.

7 Limitations

There are inevitably a number of limitations with our study, some of which are specific to our chosen research methodology. We discuss the limitations and the steps we took to offset them.

Throughout our study, the tool researchers and designers were actively involved in the development and evaluation of the tool. We recognize that this active role of the researcher may have positively influenced the attitudes of the developers and managers towards the tool in the survey and follow-up questions. It even may have changed how they used the tool. This limitation is an artifact of our design science research methodology as the role *"of the researcher is central and desirable, rather than being a dismaying incursion of subjectivity that is a threat to validity."* Sedlmair et al. (2012) Indeed, the close knowledge of the teams and their needs informed the design of the tool so that it would solve their specific problems, and it influenced the nature of the questions asked and how they were phrased in the survey and interviews. This contextual knowledge was also instrumental in the analysis of the responses we received.

The developers might have been subject to the "social desirability bias" when answering questions in the interviews. Therefore, they might have exaggerated when reporting how often they use the tool and how useful it is. The novelty of the tool and some of the graphical aspects of the user interface are also confounding factors to take into account. Several developers expressed the fact that *Themis* was "prettier" than other tools as one of the reasons why they would use it. Although participants clearly indicated that the individual feedback was essential, regardless of its graphical presentation, this is a mitigating factor that needs to be kept in mind when evaluating our findings.

We acknowledge that only 8 out of 14 developers answered our survey in the first case study, and only 6 out of 40 developers participated in our interviews in the second case study. We recognize that other developers may have a different experience and that developers that did not like the tool features may not have participated in our studies. However, the findings we report are consistent with the managers' point of view of their developers' opinions on *Themis*.

Finally, we hope that the description of the tool given in this paper is sufficiently detailed should other researchers wish to implement and evaluate a similar system. Additional screenshots from *Themis* are available online.[7]

8 Conclusion and Future Work

Organizations that own legacy software systems aim to improve the quality of their source code. To that extent they may focus on two quality metrics: the number of code smells and the level of test coverage. Based on these metrics they further promote their developers to

[7] http://promyze.com/themis

follow good coding practices such as *avoiding any new code smell* or *reaching a 85% code coverage*.

Facing some difficulties, two such organizations (Pôle Emploi and Sopra Steria) asked us to help them to better engage their developers in following these practices. During 2 years we then worked with them to develop an approach that lies on two well known concepts: *Individual Feedback* and *Gamification*.

Our case study shows that *individual feedback* provides a true added value to the developers as it motivates them in an actionable manner to improve the quality of their code. For instance when a developer receives an individual feedback explaining that a violation has just been created, the developer is naturally encouraged to fix it.

Regarding *gamification*, our case study that such a concept may bring fun and then motivate the developers to take a better attention to software quality. However, even if the use of *gamification* in software engineering is gaining momentum, our study also shows that not all developers want to join in a game, and therefore forcing gamification introduces a lot of risks. Our study further shows that game design has a huge impact on the benefits that gamification may bring. We then feel that much more research is needed into the benefits and risks of gamification, while at the same time there are more tasks in software engineering where gamification could be introduced.

Further, we learnt that deploying *individual feedback* and *gamification* within a company raises some concerns regarding scrutiny. The developers fear their managers to use these concepts for management purposes (i.e. pay increase). We therefore recommend the manager to clearly express the fact that they won't use these mechanisms for performance monitoring. Finally, we also observed that these concepts require to be organized, and then recommend a developer to play that role of organizer.

As future work, we plan to conduct a *longitudinal study*, integrating quantitative elements such as usage statistics or project-level metrics, of individual feedback and gamification use, which could provide insights about how these mechanisms may be used over time and the impact of them on long-term code quality. Will these mechanisms succeed in motivating developers to follow good coding practices over one or two years of a project, or will the novelty wear off? Additionally, are there adverse effects of using individual feedback or gamification? Does it induces a loss of "collective" code knowledge as developers only see feedback about their own code? Does it foster bad developer behavior as there might be some developers that prefer earning badges than performing pending tasks?

We further agree with Pedreira et al. (2015) that there is a need for *comparative studies* and that we should strive to conduct studies of developers doing the same task in a gamified and a non-gamified manner, or with gamification realized in different ways. However, we note that doing so is very difficult due to many possible confounding factors such as age, gender, experience of the developers, team spirit and size. For instance, in our studies, most of the developers were male, between the age of 25-34, and many on the team already played games. The success of gamification in our context may have been in part due to age, as Dorling et al. note that Generation Y users appreciate clear goals, trackable progress, and social rewards (Dorling and McCaffery 2012). In terms of gender, Gneezy et al. (2003) and Vasilescu (2014) found some differences in how females participate within the gamified Stack Overflow environment. With a larger sample of participants, we may see differences in how females respond to the gamification of good coding practices. *Personality* is another consideration, as one manager noted: *"Some developers have a more discreet nature and gamification may not be a good motivator for these ones"* [m_2].

In the meantime, we hope that our findings from this study will prove useful to both researchers and practitioners interested in the role of gamification in software engineering and how individual feedback can motivate developers to follow good coding practices.

Acknowledgements We thank our research participants and Cassandra Petrachenko for improving our paper.

References

Anderson JR, Corbett AT, Koedinger KR, Pelletier R (1996) Cognitive tutors: lessons learned. https://hal.archives-ouvertes.fr/hal-00699789, aRI Research Note 96-66

Andrews JH, Briand LC, Labiche Y, Namin AS (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans Softw Eng 32(8):608–624. https://doi.org/10.1109/TSE.2006.83

Arai S, Sakamoto K, Washizaki H, Fukazawa Y (2014) A Gamified tool for motivating developers to remove warnings of bug pattern tools. In: Proceedings of the 2014 6th international workshop on empirical software engineering in practice, IEEE Computer Society, Washington, DC, USA, IWESEP '14, pp 37–42. https://doi.org/10.1109/IWESEP.2014.17

Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W (2008) Using static analysis to find bugs. IEEE Softw 25(5):22–29. https://doi.org/10.1109/MS.2008.130

Azevedo R, Bernard RM (1995) A meta-analysis of the effects of feedback in computer-based instruction. J Educ Comput Res 13(2):111–127. https://doi.org/10.2190/9LMD-3U28-3A0G-FTQT

Barik T, Murphy-Hill E, Zimmermann T (2016) A perspective on blending programming environments and games: beyond points, badges, and leaderboards. In: 2016 IEEE symposium on visual languages and human-centric computing (VL/HCC), pp 134–142. https://doi.org/10.1109/VLHCC.2016.7739676

Beck K, Fowler M, Beck G (1999) Bad smells in code. Refactoring: Improving the design of existing code, pp 75–88, http://www-public.tem-tsp.eu/gibson/Teaching/Teaching-ReadingMaterial/BeckFowler99.pdf

Beecham S, Baddoo N, Hall T, Robinson H, Sharp H (2008) Motivation in Software Engineering: a systematic literature review. Inf Softw Technol 50(9–10):860–878. https://doi.org/10.1016/j.infsof.2007.09.004. http://www.sciencedirect.com/science/article/pii/S0950584907001097

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: a large-scale evaluation in open source software. In: 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER), vol 1, pp 470-481. https://doi.org/10.1109/SANER.2016.105

Bennett KH (1995) Legacy systems: coping with success. IEEE Softw 12:19–23

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ACM, New York, NY, USA, ASE 2016, pp 332–343. https://doi.org/10.1145/2970276.2970347. http://doi.acm.org/10.1145/2970276.2970347

Curtis B, Sappidi J, Szynkarski A (2012) Estimating the principal of an application's technical debt. IEEE Software 29(6):34–42. http://doi.ieeecomputersociety.org/10.1109/MS.2012.156

Dal Sasso T, Mocci A, Lanza M, Mastrodicasa E (2017) How to gamify software engineering. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), IEEE, pp 261–271. http://ieeexplore.ieee.org/abstract/document/7884627/

Deci EL, Koestner R, Ryan RM (1999) A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. Psychol Bull 125(6):627–668. https://doi.org/10.1037/0033-2909.125.6.627

Deterding S, Dixon D, Khaled R, Nacke L (2011) From game design elements to gamefulness: defining "Gamification". In: Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, ACM, New York, NY, USA, MindTrek '11, pp 9–15. https://doi.org/10.1145/2181037.2181040, http://doi.acm.org/10.1145/2181037.2181040

Dorling A, McCaffery F (2012) The gamification of SPICE. In: International conference on software process improvement and capability determination. Springer, Berlin, pp 295–301

Fjóla Tómasdóttir K, Finavaro Aniche M, van Deursen A (2017) Why and how JavaScript developers use linters. In: 32nd IEEE/ACM international conference on automated software engineering

França ACC, Gouveia TB, Santos PCF, Santana CA, FQBd Silva (2011) Motivation in software engineering: a systematic review update. In: 15Th annual conference on evaluation assessment in software engineering (EASE 2011), pp 154-163. https://doi.org/10.1049/ic.2011.0019

Gneezy U, Niederle M, Rustichini A et al (2003) Performance in competitive environments: gender differences. Quarterly Journal of Economics-Cambridge Massachusetts 118(3):1049–1074

Hall T, Sharp H, Beecham S, Baddoo N, Robinson H (2008) What do we know about developer motivation? IEEE software 25(4):92

Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th international conference on software engineering, ACM, New York, NY, USA, ICSE 2014, pp 435–445. https://doi.org/10.1145/2568225.2568271. http://doi.acm.org/10.1145/2568225.2568271

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 international conference on software engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

Letouzey JL, Ilkiewicz M (2012) Managing technical debt with the SQALE method. IEEE Softw 29(6):44–51. https://doi.org/10.1109/MS.2012.129

Miller JC, Maloney CJ (1963) Systematic mistake analysis of digital computer programs. Commun ACM 6(2):58–63. https://doi.org/10.1145/366246.366248. http://doi.acm.org/10.1145/366246.366248

Mockus A, Nagappan N, Dinh-Trong TT (2009) Test coverage and post-verification defects: a multiple case study. In: 2009 3rd international symposium on empirical software engineering and measurement, pp 291–301. https://doi.org/10.1109/ESEM.2009.5315981

Nadler DA (1979) The effects of feedback on task group behavior: a review of the experimental research. Organ Behav Hum Perform 23(3):309–338

Pedreira O, García F, Brisaboa NR, Piattini M (2015) Gamification in software engineering - a systematic mapping. Inf Softw Technol 57:157–168. https://doi.org/10.1016/j.infsof.2014.08.007

Prause CR, Jarke M (2015) Gamification for enforcing coding conventions. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 649–660. https://doi.org/10.1145/2786805.2786806. http://doi.acm.org/10.1145/2786805.2786806

Rojas JM, White TD, Clegg BS, Fraser G (2017) Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game. In: Proceedings of the 39th international conference on software engineering, IEEE Press, Piscataway, NJ, USA, ICSE '17, pp 677–688. https://doi.org/10.1109/ICSE.2017.68

Schooler LJ, Anderson JR (1990) The disruptive potential of immediate feedback. In: Proceedings of the twelfth annual conference of the Cognitive Science Society, pp 702–708

Seaborn K, Fels DI (2015) Gamification in theory and action: a survey. Int J Hum Comput Stud 74:14–31. https://doi.org/10.1016/j.ijhcs.2014.09.006. http://www.sciencedirect.com/science/article/pii/S1071581914001256

Sedlmair M, Meyer M, Munzner T (2012) Design study methodology: Reflections from the trenches and the stacks. IEEE Trans Vis Comput Graph 18(12):2431–2440

Singer L, Schneider K (2012) It was a bit of a race: gamification of version control. In: Proceedings of the second international workshop on games and software engineering: realizing user engagement with game engineering techniques, IEEE Press, Piscataway, NJ, USA, GAS'12, pp 5–8. http://dl.acm.org/citation.cfm?id=2663700.2663702

Snipes W, Nair AR, Murphy-Hill E (2014) Experiences gamifying developer adoption of practices and tools. In: Companion proceedings of the 36th international conference on software engineering, ACM, New York, NY, USA, ICSE Companion 2014, pp 105–114. https://doi.org/10.1145/2591062.2591171. http://doi.acm.org/10.1145/2591062.2591171

Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media

Tikir MM, Hollingsworth JK (2002) Efficient instrumentation for code coverage testing. SIGSOFT Softw Eng Notes 27(4):86–96. https://doi.org/10.1145/566171.566186. http://doi.acm.org/10.1145/566171.566186

Vasilescu B (2014) Human aspects, gamification, and social media in collaborative software engineering. In: Companion proceedings of the 36th international conference on software engineering, ACM, pp 646–649

Williams TW, Mercer MR, Mucha JP, Kapur R (2001) Code coverage, what does it mean in terms of quality? In: Annual reliability and maintainability symposium. 2001 Proceedings. International symposium on product quality and integrity (Cat. No.01CH37179), pp 420–424. https://doi.org/10.1109/RAMS.2001.902502

Matthieu Foucault is a postdoctoral fellow at the University of Victoria. He received his Ph.D. degree from the University of Bordeaux. His research interests include software evolution and data visualization.

Xavier Blanc is full professor at the Bordeaux University. He holds a Research Direction Habilitation in Computer Science from Paris 6 University in 2009. His current research is about software evolution. He works on repository mining and on static analysis.

Jean-Rémy Falleri is associate professor at the Enseirb-Matmeca french engineering school and researcher at the LaBRI laboratory. He obtained his habilitation in 2015 at the Université de Bordeaux. He received the Ph.D degree from University of Montpellier. His research interest includes software engineering and software evolution.

Margaret-Anne Storey is a Professor of Computer Science and a Co-Director of the Matrix Institute for Applied Data Science at the University of Victoria, and was Director of the Software Engineering Program from 2015 to 2018.

Dr. Storey's research goal is to understand how technology can help people explore, understand, and share complex information and knowledge.