Published by Elsevier
DOI: https://doi.org/10.1016/j.infsof.2022.106946
Accepted Manuscript: INFSOF-D-21-00519R2

The role of awareness and gamification on technical debt management

Yania Crespo González-Carvajal[1], Carlos López-Nozal[2], Raúl Marticorena-Sánchez[2], Margarita Gonzalo Tasis[1], and Mario Piattini[3]

[1] Departamento de Informática, Universidad de Valladolid, España
{yania,marga}@infor.uva.es
[2] Departamento de Ingeniería Informática, Universidad de Burgos, España
{clopezno,rmartico}@ubu.es
[3] Departamento de Tecnologías y Sistemas de Información, Universidad de Castilla La Mancha, España
Mario.Piattini@uclm.es

Abstract. *Context.* Managing technical debt and developing easy-to-maintain software are very important aspects for technological companies. Integrated development environments (IDEs) and static measurement and analysis tools are used for this purpose. Meanwhile, gamification also is gaining popularity in professional settings, particularly in software development. *Objective.* This paper aims to analyse the improvement in technical debt indicators due to the use of techniques to raise developers' awareness of technical debt and the introduction of gamification into technical debt management. *Method.* A quasi-experiment that manipulates a training environment with three different treatments was conducted. The first treatment was based on training in the concept of technical debt, bad smells and refactoring, while using multiple plugins in IDEs to obtain reports on quality indicators of both the code and the tests. The second treatment was based on enriching previous training with the use of `SonarQube` to continuously raise awareness of technical debt. The third was based on adding a gamification component to technical debt management based on a contest with a top ten ranking. The results of the first treatment are compared with the use of `SonarQube` for continuously raising developers' awareness of technical debt; while the possible effect of gamification is compared with the results of the previous treatment. *Results.* It was observed that continuously raising awareness using a technical debt management tool, such as `SonarQube`, significantly improves the technical debt indicators of the code developed by the participants versus using multiple code and test quality checking tools. On the other hand, incorporating some kind of competition between developers by defining a contest and creating a ranking does not bring about any significant differences in the technical debt indicators. *Conclusions.* Investment in staff training through tools to raise developers' awareness of technical debt and incorporating it into continuous integration pipelines does bring improvements in technical debt management.

Keywords: Technical Debt · Raising awareness · SonarQube · Gamification · Quasi-experiment

1 Introduction

Current and future Software Engineering professionals must be trained in techniques and tools that allow them to speed up the production of quality software, using the best practices in software development, particularly in those techniques and tools that keep Technical Debt (TD) under control and reduce it.

In 1992, Ward Cunningham coined the term "Technical Debt" [16], based on the concept of financial debt, as the main factor that makes software project development and maintenance slow and laborious. Cunningham himself relaunched the concept in a series of keynote conferences around the world in 2008, and since 2010 the Managing Technical Debt workshop series has provided a forum for practitioners and researchers to discuss issues related to TD. In 2016, a Dagstuhl Seminar on "Managing Technical Debt in Software Engineering" produced a consensus definition for TD, a draft conceptual model, and a research roadmap [6]. Its definition states that "technical debt is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible". The First International Conference on Technical Debt, held in 2018, started a new series of conferences as the successor to the previous Managing Technical Debt workshop series. TD is still an element of undoubted interest today, since, as published in the State of Technical Debt report in 2021 [49], software developers nowadays waste around one day a week on paying TD.

Understanding, communicating, and managing TD can make a huge difference in both the short and long term success of a system [3, 6]. The software engineering community is coming together on defining TD as making technical compromises between the short and the long term. Initially, the introduction of TD was considered a deliberate action. Nowadays, it is assumed that, on many occasions, the introduction of TD is not intentional and the novice developer plays a fundamental role in this issue. Hence the importance of training future software development professionals to recognise, analyse, monitor and measure TD in order to be aware of and reduce unintentional TD. Falling into TD within a team of expert developers is inevitable, despite the adoption of prudent strategies in development. However, a novice team ignorant of design practices may take on its reckless debt without even realizing how much effort it will have to make in the future. This is why it may be interesting to bring to the attention of novice developers concrete information about TD that will help them to be more prudent and practise an aware self-management.

It is, first, a way to explain the balance between immediate delivery vs. long-term maintainability, because it allows its integration and interaction with tools, the estimation of models for decision-making, facilitates the understanding of the internal quality of software and can be applied to generating economic models.

Providing incentives may help in reducing technical debt and gamification is such a strategy for incentives. Gamification seeks to improve motivation and efficiency when carrying out certain tasks by incorporating game mechanisms and elements to make the task more attractive [40, 48]. It has already been pointed out that an incentive strategy may be appropriate for the improvement of TD, and gamification seems to be a suitable alternative [11, 35].

Based on this background, we investigated the extent to which awareness-raising and/or gamification of TD management has an impact on TD indicators. To this end, we conducted a quasi-experiment [55] with a comparative analysis between three different training environments for undergraduate developers and the influence of some factors on some TD indicators of the developed code. In particular, the role of continuously raising awareness of TD by means of `SonarQube` integrated in a CI/CD pipeline and the introduction of gamification strategies related to these indicators are analysed and compared to more classical contexts, such as using multiple plugins in IDEs to obtain reports on code quality indicators and tests.

This paper is therefore organised as follows: the background concerning TD, the use of `SonarQube` and the inclusion of gamification is presented in Sec. 2. The quasi-experiment is presented in Sec. 3, setting out the study design. The results obtained in response to our research questions are then detailed in Sec. 4, followed by a discussion of these results in Sec. 5. In Sec. 6, we review possible threats to the validity of our study and finally, in Sec. 7, we present the fundamental conclusions and lines of future work.

2 Related Work

Fowler [25] divides TD into two dimensions: reckless/prudent and deliberate/inadvertent. Our work focuses more on inadvertent, mainly reckless TD. Another classification of the different types of TD can be found in Rios et al. [44], the most referenced ones being: design debt (debt that can be discovered by analysing the source code and identifying violations of the principles of good object-oriented design), code debt (poorly written code that violates best coding practices or rules), test debt (issues found in testing activities that can affect the quality of the product), and documentation debt (insufficient, incomplete or outdated documentation). We focused on these four which are also closely related to maintainability indicators.

2.1 Software Quality and Technical Debt awareness

Static analysis of the source code to help improve TD has been studied in several works. `SonarQube` is mainly used as the reference tool [17], and it is combined with others such as `GitHub` and `Microsoft Project` [42], `CodeClimate` [51], and `Findbugs` [38]. How the use of tools increases motivation to improve TD is mostly studied with satisfaction surveys and qualitative analyses [17, 42], but

without cross-checking with the final data on TD or statistical validation [17, 41].

Improvements in students' TD has been checked by studying projects developed by students [17, 38, 41, 42, 51]. However, only the final results have been observed [41], without studying the effect of raising awareness.

In relation to the more explicit awareness of TD, work has been done to introduce changes over successive courses [17, 51]. As can be deduced from the results, prior information and awareness-raising work, combined with the use of automatic measurement tools, had a positive influence on reducing debt and improving the quality of the software. However, these studies focus their conclusions only on the analysis of survey data. Regarding automatic reviews with tools in training environments, the effectiveness of quality tools used in the evaluation of projects in Software Engineering courses was investigated in Silva et al. [47]. This paper concluded that students receive solid feedback through the use of these tools. Ramasubbu & Kemerer [43] also proposed a normative framework for the integration of TD management and software quality management. As a practical example, in Liu & Woo [32], `SonarQube` was used to provide code quality to an automatic online evaluation system. The authors indicated that the significant advantages of the system were that it helps the instructor to discover weaknesses during a lecture and the students to locate their mistakes efficiently.

The application of different testing techniques on different groups (non-testing, traditional and TDD, respectively), combined with the use of such tools as `Findbugs` or `SonarQube` to measure TD, reveal that the results are not always as expected, with hardly any reduction in TD when applying traditional or TDD testing [38].

2.2 Gamification and Technical Debt

Gamification in Software Engineering has been studied as a mechanism to improve engagement and job satisfaction [50] and has been applied to obtaining better code quality [39].

The reduction of TD with incentives has been studied in other works [11], but not from a purely gamification perspective. The problem of the diversity of evaluation in software engineering gamification [35] suggests that its application to TD could be an option. While the evaluation in most cases was subjective and qualitative, its combination with TD provides a more objective quantitative analysis.

The interest of empirical studies in the effectiveness of gamification in Software Engineering education has been highlighted [1], especially as an innovative effect of interest, which can improve motivation in software development, but the lack of empirical studies with quantitative results has been noted [2].

Nguyen & Bodden [36] propose including gamification features similar to those included in video games. A wireframe prototype of a static code analysis tool enriched with gamification elements was used to survey 8 experts. They concluded that it is convenient to gamify these tools with elements such as points, badges or profiles as they help the programmer's commitment. However,

without a quantitative experimental study with a group of trainees, and without working on real projects or programming tasks, the results were limited to the opinions gathered from the panel of experts.

Theoretical approaches introducing gamification from a serious games perspective on TD, together with refactoring, testing and the use of quality analysis for the calculation of TD, can be found in Haendler et al [27]. More applied approaches can also be found, such as the work by Atal et al [5]. In this case, the authors introduce gamification in teaching peer code reviewing in order to positively affect code quality and TD. The empirical methods used are limited to surveying participants.

The *Themis* tool [23] was proposed, integrating TD support, version control and gamification, in order to improve TD management. The study analysed the application of this tool to a single project already in operation, with a team of 3 managers and 14 developers. The study method is based on a survey and subsequent qualitative evaluation of the opinions collected, but no quantitative results in terms of TD reduction are analysed.

Some empirical research has been conducted [10] by defining a quadrant of different TD management strategies, such as encouraging, rewarding, penalising and forcing. The results, obtained from surveys and interviews, suggest that developers were not usually rewarded, penalised, or forced in their companies to maintain low levels of TD. 60% of the respondents stated that they were encouraged, to some extent, to keep the level of TD down. An extended work [11] emphasises incentives (rewarding and encouraging). Regarding the encouraging approach, the introduction of an additional element such as gamification seems to be promising.

Inspired by Besker et al. [10], we revisited the strategies these authors defined, but in an educational setting [14]. The encouraging strategy is the most usual in educational contexts, with instructors always encouraging students to keep TD low. The force strategy in an educational context prevents the evaluation of student performance and learning, and can lead to a total lack of motivation. Therefore, we focused on comparing the rewarding and penalising strategies, which in our case corresponds to giving teams formed by 3 or 4 students in our "Software Architecture and Design" course either incentives or disincentives. The study concluded that the reward strategy (the carrot), based on gamification, worked better than the penalty strategy (the stick), based on quality gates, to motivate the students to keep TD low and produce a high quality code.

The study we present in this paper differs from previous works in the following points: (a) different product metrics related to TD indicators are studied, (b) not only a descriptive study is carried out, but also a quantitative analysis with statistically significant results to draw objective conclusions, and (c) the effect of raising developers' awareness when faced with the introduction of gamification is analysed.

3 Study Design

We performed a quasi-experiment following the recommendations provided by Kitchenham et al. [29] and Wohlin et al. [55].

In the following, we first define the goal of our quasi-experiment and present the research questions (Section 3.1). We then present the independent and dependent variables (Section 3.2) and formulate the hypotheses defined to investigate the research questions (Section 3.3). We provide details on the participants in the experiment (Section 3.4), present its design (Section 3.5) and the infrastructure used to collect and analyse data (Section 3.6). The section concludes by highlighting the experiment's operation (Section 3.7).

3.1 Goal and Research Questions

The main goal of the quasi-experiment is defined by applying the Goal Question Metrics (GQM) template [8] as follows:

Analyse the impact of continuously raising developers' awareness on TD and the impact of gamification on TD management
for the purpose of evaluating their effects
with respect to the source code quality
from the point of view of the researchers
in the context of a third year software engineering course involving undergraduate students.

The research questions we addressed are the following:

RQ1 *Does raising developers' awareness of technical debt improve technical debt indicators?*
RQ2 *Does introducing gamification in technical debt management improve technical debt indicators?*

3.2 Selection of Variables

The **independent variable** represents the training environment. Three different treatments were conducted depending on the training environment. The first treatment was based on training the theoretical concept of TD, bad smells and refactoring, while using multiple plugins in IDEs to obtain reports on quality indicators of the code and the tests in a more traditional way. This treatment is labelled as core (see Sec. 3.5.1 for its description).

The second treatment was based on enriching previous training with the use of `SonarQube` to continuously raise awareness of TD by means of a CI/CD pipeline. This treatment is labelled as sonarqube (see Sec. 3.5.2 for its description).

The third treatment was based on adding a gamification component to TD management. This add-on was based on the definition of a contest with a top

ten ranking. This treatment is labelled as gamification (See Sec. 3.5.3 for its description).

Therefore, the independent variable is a nominal variable and takes three values: core, sonarqube and gamification.

The dependent variables of the quasi-experiment are metrics obtained from the code developed by the subjects to accomplish the assigned programming task.

The **dependent variables** considered in this study are:

- *smells density* (SD),
- *technical debt ratio* (TDR),
- *branch_ coverage* (BC),
- *code to test* ($C2T$),
- *comment density* (CD).

The first two are related to the design and code debt, the second two to test debt and the last one to documentation debt (see background in Sec. 2). Ratio scale metrics have been selected to avoid limitations in the use of resources to assess the observed data. These metrics are defined below.

3.2.1 Design and Code debt related metrics

Smells Density (SD) It is important to detect the presence of *Code Smells* to evaluate the design and code debt. Therefore, the metric *code smells* is selected, which calculates the total amount of smells present in the code. From this absolute amount, a derived metric named smells density is defined as a rate as follows:

SD = code smells/lines of code.

Technical Debt Ratio (TDR) Another metric related to design and code debt included in the study is *technical debt ratio*. It is defined in `SonarQube` as the ratio between the cost of developing the software and the cost of fixing it:
Remediation cost / Development cost
This can be restated as:

*TDR = Remediation cost / (Cost to develop 1 line of code * Number of lines of code)*

The value of the cost of developing a line of code is considered to be 0.06 days by default in `SonarQube`.

3.2.2 Test debt related metrics

Two types of metrics are used related to test debt: static vs. dynamic, i.e., those which can either be obtained by static analysis or those that require an execution to obtain them.

Coverage and test success metrics are dynamic metrics. They are calculated by performing an execution of the code to be tested. There are several types of coverage metrics and different tools to obtain them. Coverage metrics are percentages which indicate the code that the tests execute against the total code that could be executed. It is usual to take into account the metrics *line coverage* and *branch coverage*, or even *coverage* which is calculated by combining the two previous metrics. The metric *line coverage* is not interesting in this study because it is very easy to obtain high levels of this kind of coverage in small size projects. We therefore include *branch coverage* in the study, which is the metric that could make the difference.

Branch Coverage (BC) On each line of code containing some boolean expressions, the branch coverage simply answers the following question: 'Has each boolean expression been evaluated as both true and false?' This is the density of possible conditions in flow control structures that have been followed during the execution unit tests.

*BC = (CT + CF) / (2*B)* where:

CT = the conditions that have been evaluated as 'true' at least once;
CF = the conditions that have been evaluated as 'false' at least once;
B = the total number of conditions.

Code to Test (C2T) This is a static metric, so it is calculated by the static analysis of both the main code or production code and the test code [26]. In this work, we calculate this metric as a rate between the number of unit tests (tests) and the lines of production code. It is obtained from the metrics *tests* and *lines of code* obtained with `SonarQube`:

C2T=tests/lines of code.

3.2.3 Documentation debt related metric

The participants were instructed in the principle "When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous." [24]; which is similar to "Don't Use a Comment When You Can Use a Function". According to this principle, they were also instructed to only use javadoc comments in the code. We verified that all comments in the code developed by the participants were javadoc comments. The quality of the javadoc comments were evaluated in order to improve understandability and thus maintainability. Hence, the number of comment lines (metric *comment lines*) was considered an interesting indicator of Documentation debt.

Comment Density (CD) is a derived metric defined as a rate as follows:

CD = Comment lines / (Lines of code + Comment lines).

3.3 Hypotheses formulation

Considering the related work, raising awareness and introducing gamification on TD management is expected to have positive effects on the quality of software code. Therefore, we formulated the following hypotheses:

H_1^{RQ1}: TD indicators regarding design and code debt, test debt and documentation debt behave better on average when developers are trained by continuously raising awareness of TD (sonarqube treatment) than when trained with core treatment.

H_1^{RQ2}: TD indicators regarding design and code debt, test debt and documentation debt behave better on average when developers are trained adding gamification of TD management (gamification treatment) than when trained with sonarqube treatment.

3.4 Sampling and participants

Over a period of several years, we have worked on the continuous improvement of training environments for Software Engineering professionals in terms of quality software development and TD management.

The target population of this study are novice software developers. Due to the availability of the subjects and the possibility of controlling the study, a quasi-experiment has been carried out in the context of the University of Valladolid. It is a quasi-experiment (according to Wohlin [55]) as the assignment of treatments to subjects is a human-oriented investigation and cannot be based on randomisation, but emerges from the characteristics of the subjects.

Focusing on the intended population of novice software developers, the participating subjects were full-time third year students of a major in Software Engineering, voluntarily enrolled in an optional course called *Technologies for Software Development* in three different academic years (2016-2017, 2017-2018, 2018-2019), giving a total of 77 participants without repetition. The sample takes students who are about to enter the labour market the following year, with a uniform profile of knowledge and previous experience, and coincides with that of a novice programmer.

According to the definition of the quasi-experiment , we have three groups, 25 subjects participated in the first treatment (core) of which 2 were female and 23 were male, 24 subjects in the second treatment (sonarqube) of which 1 was female and 23 were male, and 28 in the third (gamification) all of whom are male. The participants in the three groups were 21 years old on average, had no previous work experience in Software Engineering and the same previous

academic experience. The preceding subjects they had taken were the same for all and did not change from one group to another.

Unfortunately, as has been common in the Software Engineering profession and studies, there were hardly any female members in the groups.

We have analysed the three groups by studying the grades obtained by the participants in each group in the 20 subjects of the first two courses, finding no significant differences between the groups in any of the subjects.

3.5 Experiment design

A package of 150 hours of training (60h theoretical + 90h practical) was defined. This package includes framework-based development, domain-specific languages, agile practices, pair programming, Test First, the complete TDD cycle (*Red*-*Green*-*Refactor*), different testing levels automation, isolation testing using mocks, coverage and test monitoring, self-documentation, configuration management (in particular version control and issue tracking), project and dependency management automation, continuous integration and continuous delivery (CI/CD), bad smells in code, and refactoring. In addition, the concept of TD and different metrics and indicators related to code quality and TD management were also introduced.

The practical part of the training was carried out through the development of four programming tasks as assignments, identified as p1, p2, p3, and p4, applying the knowledge and tools of the training package. The relevant programming tasks for this work were p3 and p4:

- p3 consisted of a programming task developed by *pair programming*, completing the TDD cycle (*Red*-*Green*-*Refactor*) starting from the *Green* phase (since the *Red* phase, test first, was performed in the previous assignment p2). In this study, p2 is subsumed into p3.
- p4 is developed individually, completing the whole TDD cycle, *Red*-*Green*-*Refactor*.

A detailed description of the programming task to be developed was given to the students. Detailed instructions were also given. The instructions contain an explanation of the TDD cycle and indications for approaching it. Appendix A contains the description of each programming task (Sec. A.1), and the instructions given for each assignment p3/p4 at each treatment, presented through the differences (Sec. A.2).

In terms of the time spent, the programming tasks were around 20 hours of work. The programming tasks were carefully designed to be different but comparable in terms of size and difficulty in achieving high levels of coverage. The size metrics *lines of code* and *classes* (number of classes), and the metric *cyclomatic complexity*, are relevant to measure the number of different paths from input to output, i.e., counting these paths and *branches to cover*, which counts the conditions to be covered by the tests. The programming tasks were designed to be solved by implementing 2 or 3 classes, between 300 and 400 lines

Table 1. Subjects and objects per training treatment in the quasi-experiment

Treatment	Subjects	Projs. p3	Projs. p4	Tools	Contents and Practices
core	25	12	25	`git`, `GitLab`, `Bitbucket` (mirror), `ant`, `maven`, Eclipse+plugins: EclEmma, Eclipse Metrics, Eclipse Refactoring `pmd`	Continuous integration and automation are emphasised. Technical Debt concept is introduced. A target "measures" with `ant` is explicitly required in the project assignments.
sonarqube	24	12	19	idem previous + `SonarQube` (includes `pmd` as plugin)	idem previous + Technical Debt concept is linked to SQALE framework. The `ant` target "measures" is adapted to run the analysis with `SonarQube` and integrated in the gitlab CI/CD pipeline. Continuously raising awareness of Technical Debt indicators with `SonarQube`.
gamification	28	12	16	idem previous	idem previous + gamification (contest and ranking)
Total	**77**	**36**	**60**		
		96			

of code, and from 50 to 100 for the aggregated cyclomatic complexity (depending on the programming style).

As mentioned before, p3 was implemented by pair programming, while p4 was individual. p4 was used as a substitute for the exam. The participants voluntarily applied for this modality. This is the main reason why there are fewer p4 projects than subjects in the study.

Table 1 shows the summary of the subjects and objects for each training treatment in the quasi-experiment.

The risk of communication across the three groups along the quasi-experiment is minimised by assigning different programming tasks to each group. Each group studied in a different year and some participants had already graduated, while others were taking the course.

The risk of communication between the participants in the same group always exists but it is minimised, as in any course assignment, by penalising plagiarism.

The quasi-experiment was designed to have one factor that determines three different treatments. The first, the so-called core, was taken as the baseline. From there, it was compared with the treatment, in which monitoring and measurement TD and other related indicators were systematically performed using `SonarQube`. This was, in turn, compared with the treatment in which the gamifi-

Table 2. Quasi-experiment design: three training treatments (core, sonarqube and gamification) apply to three groups of subjects on two different objects (p3, p4).

Group	p3 Pair Programming	p4 Individual Programming
G 2016-17	core	core
G 2017-18	sonarqube	sonarqube
G 2018-19	—	gamification

cation of these indicators was introduced. The following subsections detail these three training treatments identified as **core**, **sonarqube**, and **gamification**, respectively.

Other factors are kept under control: the same instructor was involved in all three groups and the participants had the same previous acquired knowledge as presented in Sec. 3.4.

Table 2 shows the experimental configuration of the quasi-experiment, asigning each treatment to each group of participants. A detailed description of each training treatment follows.

3.5.1 The core training treatment

After several years of experience and continuous improvement, we reached the configuration of the training package and also a practical and evaluation environment formed by: `git` for local version control, `GitLab` as a remote repository, collaborative development environment and issue tracker, `Bitbucket` to practice automatic mirroring, Java as programming language, `ant` and `maven` for project automation and automatic dependency management, JUnit as the base element of test automation, Easymock/Mockito for mock-based isolation testing, Eclipse as IDE with several integrated plugins, such as EclEmma (Eclipse plugin based on Java Code Coverage-JaCoCo), for coverage analysis and test monitoring, Eclipse Metrics and `pmd` plugins for Eclipse to visualise metrics and indicators related to quality and TD. The Eclipse Refactoring plugin is also used to support the last step of the TDD cycle to improve the quality of the implemented and tested code.

The project automation with `ant` and `maven` should include a "measures" target to obtain a report of measures and indicators related to quality and TD.

Environment details per training treatment are explained as shown in Figure 1. This figure shows two common bases: communication and local environment (arranged vertically and horizontally, respectively). These bases group a series of elements common to all training treatments.

The vertical basis represents the agile communication environment used in all training treatment. These are communication channels in rocket.chat *on premises* that allow participants to communicate with each other and with the instructors. Rocket.chat is an open source project similar to Slack.

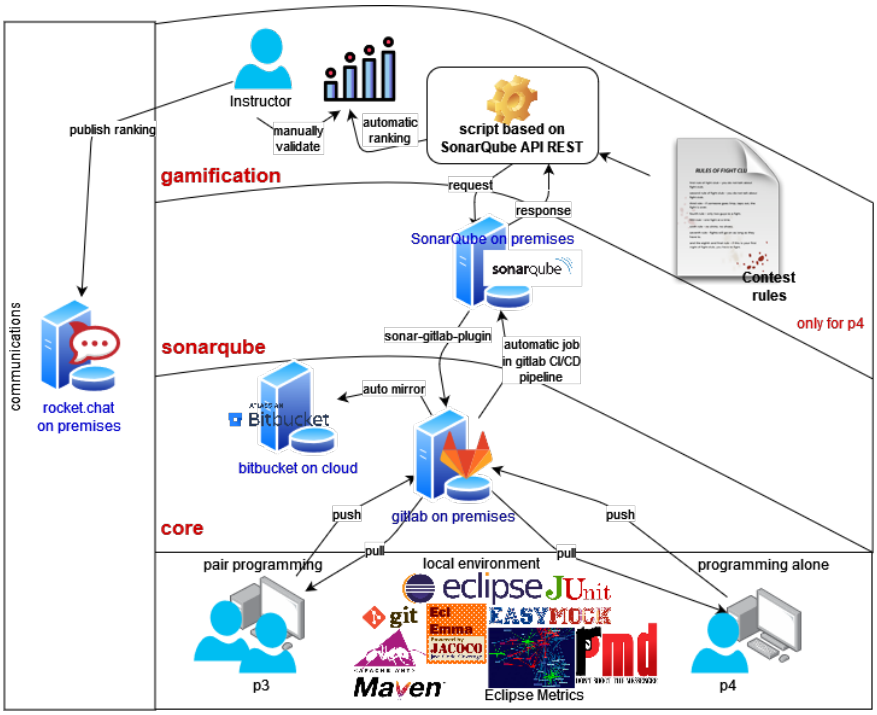

Fig. 1. Environment description per training treatment

The horizontal basis represents the local workstation environment of the participants that is common to all training treatments. This local environment, as explained above at the beginning of the description of the core training treatments, consists of `git`, `ant` and `maven`, Eclipse IDE with Refactoring, JUnit, Easymock/mockito, EclEmma (JaCoCo), Eclipse Metrics and `pmd`.

From these two common bases, three layers are depicted in Figure 1, each representing a training treatment. Each layer is stacked on top of the previous one, representing the inclusion.

The first layer represents precisely the core training treatment described in this section. This layer adds to what is represented on the horizontal basis (the local workstation), two tools that complement the local with the remote: `GitLab` *on premises* and `Bitbucket` *on cloud*.

3.5.2 The sonarqube training treatment

In the second training treatment, the use of `SonarQube` was added to the core (see second layer of Figure 1). Systematic quality control using `SonarQube`

in the p3 and p4 programming tasks was introduced. This was performed as part of a continuous integration pipeline prepared in GitLab CI/CD based on maven commands.

First, a 2-hour seminar on the use of SonarQube was introduced. Second, an explanation on how to incorporate analysis with SonarQube into the continuous integration process using GitLab CI/CD was introduced. Thirdly, in the instructions given to the participants to perform the programming task, it was indicated that the refactoring process of the TDD cycle should take into account the information obtained from SonarQube to consider refactoring opportunities. Appendix A includes the description of the instruction documents given to participants, as well as the description of the programming tasks to be developed.

Each time a participant pushes his/her project to the remote repository in GitLab, the CI/CD pipeline launches an analysis to the SonarQube server. Participants can then look at the results on that server. Additionally, the sonar-gitlab-plugin[4] was also used, which takes the results of the analysis performed with SonarQube and annotates them in the GitLab project. Unfortunately, this plugin can no longer be used after version 7.7 of SonarQube because the "preview mode" previously offered by SonarQube has been removed. Furthermore, in the local workstation, the participants were instructed to launch the analysis of the project to the SonarQube server by means of an ant target or a maven command and browse to see the results of the analysis at any time.

3.5.3 The gamification training treatment

For the third training treatment, no changes were introduced in the contents, their sequencing, or the tools to be used. A gamification strategy based on SonarQube was defined and introduced in TD management.

This training treatment is referred to as **gamification**, and includes the above (**core** + continuously raising TD awareness using SonarQube) plus the gamification strategy in TD management.

A competition with winners and prizes was defined as a gamification strategy. Gamification occurs specifically in p4. Students participate in the contest individually by performing the full cycle *Red*-*Green*-*Refactor*.

A ranking among the participants was produced. To be included in the ranking, each participant had to obtain the following percentages or values for his/her project in the SonarQube server, defined as a *quality gate*[5]:

- Test Success (TS) = 100%
- Branch Coverage (BC) $\geq$ 95%
- Density of Duplications (DD) $\leq$ 1%
- Bugs = 0

[4] https://github.com/gabrie-allaigre/sonar-gitlab-plugin

[5] At https://docs.sonarqube.org/latest/user-guide/quality-gates/ it is defined and stated: "Quality Gates enforce a quality policy in your organization by answering one question: is my project ready for release?"

– Vulnerabilities = 0
– Technical Debt (TD) $\leq$ 5 *hours*

The thresholds in the application of the Quality Gate were selected based on the experience of previous courses (historical data) with similar programming tasks in terms of size and complexity. What we aimed to do when defining the QualityGate, which allows inclusion in the ranking, was to represent the indicators of the highest quality tasks developed by the participants of previous courses.

Once in the ranking, an order was established by assigning a score to each participant according to the formula: $3 * BC + (60 * 5 - TD) + (100 - DD)$. In the formula, 60*5 represents the maximum 5 hours of TD accumulated that are required for inclusion in the ranking (expressed in minutes), minus the accumulated TD measured by `SonarQube` in minutes. In this way, the lowest accumulated TD gets the most points. In the extreme case of TD = 0 hours, 300 points are obtained, which are balanced by the extreme case in which branch coverage of 100% is reached, another 300 points maximum can be obtained in this way. However, the other addend (100-DD) must be a number between 99 and 100, since the duplicate code density (DD) must be less than 1%. The maximum score obtained in the ranking is 700 points, and at least 384 points (corresponding to the case 95*3+0+99) must be obtained for inclusion in the ranking.

Ties are solved according to the following order:

1. Higher Code to Test rate ($C2T$).
2. Lower Technical Debt Ratio (TDR).
3. Lower Code Smell Severity (CSS) (calculated as blocking+major)/violations.
4. Reaching the state that has led to the tie earlier.

The prize consists of extra points in the final grade. The distribution of points is done with the top 10 of the ranking (if any). Following the ranking, we start by distributing 1 point to number 1, 0.9 to number 2, 0.8 to number 3 and so on, until 0.1 for number 10 is reached. When the prizes obtained are added to the winners' grades, the maximum grade cannot exceed 10 points.

The participant competes by visualising the evolution of the analysis in `SonarQube` of his/her project. First, he/she attempts to meet the criteria to enter the ranking, and then attempts to improve his/her points, without seeing what the others are doing. The ranking is shown at the end, and thus the distribution of points.

The instructors must manually review the code developed by the participants in the ranking to ensure that all the requested functionality has been developed. Once the ranking has been manually reviewed, the instructors publish it in the general communication channel on rocket.chat (see upper layer of Fig. 1).

3.6 Data collection and analysis

This section describes how the study was conducted, the automation and data collection, and the measurement acquisition.

3.6.1 Data collection

The measurements for the metrics as dependent variables were obtained with SonarQube. The SonarQube server used calculates 139 metrics, each described with name, identifier or key, and type of metric[6]. The version of the SonarQube server used for the projects analysis and metrics collection was 5.6.6, since this was the one we had installed when it was used by participants for the first time in projects p3 and p4 in the second treatment (2017-2018 academic year).

Data collection for all the projects was automated. To do this, a *pom file* was defined that responds to the maven-quickstart archetype. This archetype was the one to be used by the students when developing the p3 and p4 projects in all courses. Hence, the structure of all the projects was the same. All the necessary dependencies for all the projects throughout the three training treatments were included in the *pom file*, where we also configured all the necessary plugins to produce the test execution reports and the coverage analysis, as well as the quality analyses with SonarQube. This made it possible to treat all the projects equally and to automate the analyses, even if SonarQube had not been used by the participants, as in the case of the core training treatment.

The projects were anonymised and all the original *pom files* of all the participants' projects were replaced, using a shell script, by the prepared *pom file*, generating a unique project identifier for each one, following a pattern that identifies the course and kind of project (p3 or p4).

3.6.2 Procedures

From this point on, all automation was performed using R. From an R script, HTTP requests were launched to the SonarQube server to obtain measurements of the chosen metrics for all the projects in the study. The derived metrics defined for some dependent variables were calculated. All these values were written to csv files.

The data loading was performed by the 'reading_csv()' function of the 'readr' package [54]. For data analysis, some of the variable types were also transformed (from character to factor), so they could be properly recognised in both data visualisation and data analysis.

The data of interest, measurements of dependent variables, corresponded to percentages. Thus, for the analysis of percentage data, i.e., those whose range of variation occurs between 0 and 1, we used the beta regression [22] by using the 'betareg()' function of the 'betareg' package [15].

The 'betareg()' function fits beta regression models for rates and proportions type data, via maximum likelihood, using a parameterisation with mean (depending on a link function on the co-variables) and an accuracy parameter called phi.

[6] The documentation of the metrics is available at https://docs.sonarqube.org/latest/user-guide/metric-definitions/.

Formally introduced by Paolino in 2001 [37], beta regression differs from traditional linear regression as it models a dependent variable that follows a beta distribution, not a normal distribution. The beta distribution can be parameterised by its mean and variance similar to a normal distribution. However, unlike the normal distribution, the variance of a beta distribution is a function of its mean and a 'precision' parameter, a scaling measurement describing how tightly clustered the observed data are. Beta regression uses this particular aspect of the beta distribution to model the association of explanatory variables to changes in mean and variance simultaneously.

To explore the dataset, we used the 'summary()' function of the R base package which generates a type-dependent numerical summary for each of the variables. The 'ggplot2' package [53] was used to visualise the variability of the observations of the response (dependent) variable, along the levels of the explanatory (independent) variable 'treatment'.

The 'joint_tests()' function from the 'emmeans' package [30] was used to create the **Student's t**-type table of variance analysis (characteristic of linear regression models).

For the Student's t we formulated the null hypotheses H_0, which assume that there is no difference in the TD indicators from one treatment to another. If H_0s can be rejected, the alternative hypotheses are assumed to be true, which in this case are the study hypotheses stated in Sec. 3.3. H_0s are formulated as follows:

$\mathbf{H}_0^{RQ1}$**:** $\forall M_i \in \{SD, TDR, BC, C2T, CD\}, \mu^{M_i}_{sonarqube} = \mu^{M_i}_{core}$, representing μ behaviour on average.

$\mathbf{H}_0^{RQ2}$**:** $\forall M_i \in \{SD, TDR, BC, C2T, CD\}, \mu^{M_i}_{gamification} = \mu^{M_i}_{sonarqube}$, representing μ behaviour on average.

These null hypotheses state that the TD indicators behave the same on average, independently of the treatment. Rejecting a null hypothesis according to the p-value when performing the Student's t-test allows the corresponding hypothesis (alternative H_1 in Sec. 3.3) to be accepted. Better behaviour can be assumed according to each metric interpretation (greater is better or lower is better).

The difference in means between independent treatments was assessed with the Student's t-test, using Cohen's d to quantify the size of the difference (e.g., small, medium or large). We used the effect size package [9] to calculate and interpret the standardised difference d (Cohen's d).

R scripts and data in csv files are available in the replication package[7].

3.7 Operation

This section describes how the experimental plan was implemented.

[7] https://github.com/clopezno/gamdebt.

Table 3. Number of projects (p3 in pairs and p4 individually) finally participating in the study, broken down by the values of the independent variable 'treatment'.

Treatment	p3	p4	Total
core	11	19	**30**
sonarqube	7	16	**23**
gamification	10	16	**26**
Total	**28**	**51**	**79**

3.7.1 Training The participating subjects developed the (programming tasks) projects p3 and p4, and both the quality of the main code and the automated tests were analysed.

Once developed, the objects in the study were characterised by size and complexity metrics in order to ascertain that they are comparable projects once developed by the participants, as was intended by the design (See Sec. 3.5). It was found that there is not much deviation between the quartiles of these metrics taken in the developed projects p3 and p4 of the three training treatments. Figure 2 shows the characterisation of the programming tasks on average in terms of number of classes and lines of code obtained from each group after training.

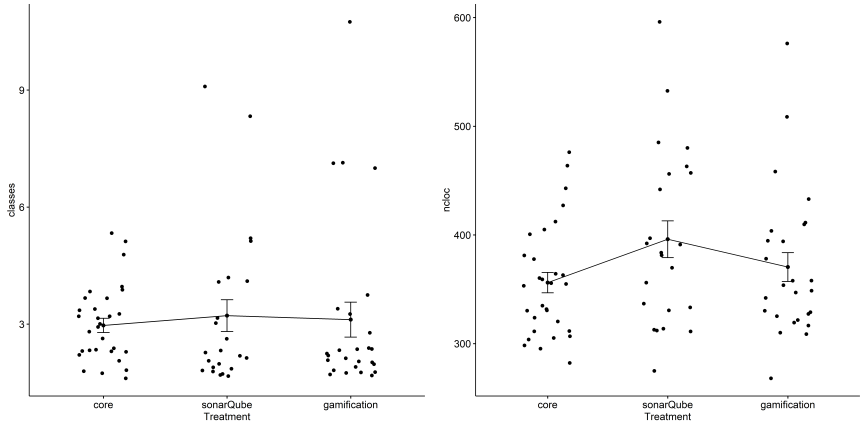

Fig. 2. Comparison of the distributions of size-related code measures (Number of classes and NCLOC) for the three treatments (core, sonarqube, gamification).

3.7.2 Experiment operation The analyses of 96 projects were launched to the `SonarQube` server. The distribution of these projects by course and kind of project is shown in Table 1. From the total number of projects, 17 were excluded,

Table 4. Quantitative analysis RQ1. p-value Student's t-test and effect size computation with Cohen's d (core vs. sonarqube).

	Smells Density (SD)	Technical Debt Ratio (TDR)	Branch Coverage (BC)	Code to Test (C2T)	Comment Density (CD)
p3 Student-t	**0.0002*** **	**0.0099** **	0.5418	**0.0461***	0.2862
effect size	**1.31 large**	**1.13 large**	-0.27 small	**-0.84 large**	0.39 small
p4 Student-t	**<.0001*** **	**<.0001*** **	0.2664	**0.0008*** **	**<.0001*** **
effect size	**2.04 large**	**2.41 large**	0.48 small	**-1.16 large**	**2.43 large**

as can be seen in Table 3, reducing the final set to 79 projects, which are the ones that finally participated in the study.

Projects were excluded for different reasons: either because they could not be analysed with `SonarQube`, because they generated errors, or because the `SonarQube` metrics could not be obtained as the tests were not available. In general, they corresponded to cases in which the participants did not complete the tasks specified in the p3 and p4 assignments. These cases could be considered as dropouts.

4 Results

This section presents the results obtained from the analysis of the dependent variables in the study, according to the procedures described in Sec. 3.6. In addition, a descriptive analysis of the data with the help of boxplot diagrams is presented. The presentation is structured around the defined research questions.

Regarding RQ1: *Does raising developers' awareness of technical debt improve technical debt indicators?*

For each metric considered, $\forall M_i \in \{SD, TDR, BC, C2T, CD\}$, and for each project (programming task) $p \in \{p3, p4\}$, we analysed the following null hypothesis:

$$H_0^{RQ1}(p) : \mu_{sonarqube}^{M_i} = \mu_{core}^{M_i}$$

Table 4 shows the results of the Student's t-tests on the two experimental scenarios considered: pair programming (p3) and individual programming (p4). The cells of the Table represent the probability of accepting the alternative hypothesis as true, when the null hypothesis could be true, known as the p-value. The p-values denoting different behaviours of the measure applying the different training treatments (core vs. sonarqube) are marked in bold according to the following degree criterion: less than 0.001 (highlighted with '***'), less than 0.01 (highlighted with '**') and less than 0.05 (highlighted with '*').

The experimental results in Table 4 suggest that the training treatment based on continuously raising awareness of the TD with the `SonarQube` tool (training

treatment core vs. training treatment sonarqube) has an impact on the metrics. Seven out of 10 of the metrics treated exhibited different behaviours regarding the means of the metrics. According to the p-values and the degree criteria, it can be seen that the influence of the training treatment is greater when the project is individual (p4), with 4 out of 5 significant values, than when it is carried out in pairs (p3), with 3 out of 5 significant values.

In addition to the detailed analysis of treatment comparison and quantification of improvement based on averages, Fig. 3 and Fig. 4 present the boxplot plots for p3 and p4, respectively, which visually corroborate the results of the detailed analysis of the considered metrics. For each project, the boxplots of the five metrics used as dependent variables are presented.

We further complement this analysis by interpreting the measurement values and analysing their improvement from the mean in the scenarios considered in our experimental design. In both scenarios, p3 and p4, Code to Test (C2T), as well as Smells Density (SD) and Technical Debt Ratio (TDR), are dependent on the training treatment of the participants. Moreover, the measurement values improve when `SonarQube` is used in training. This statement is discussed in detail for each metric below.

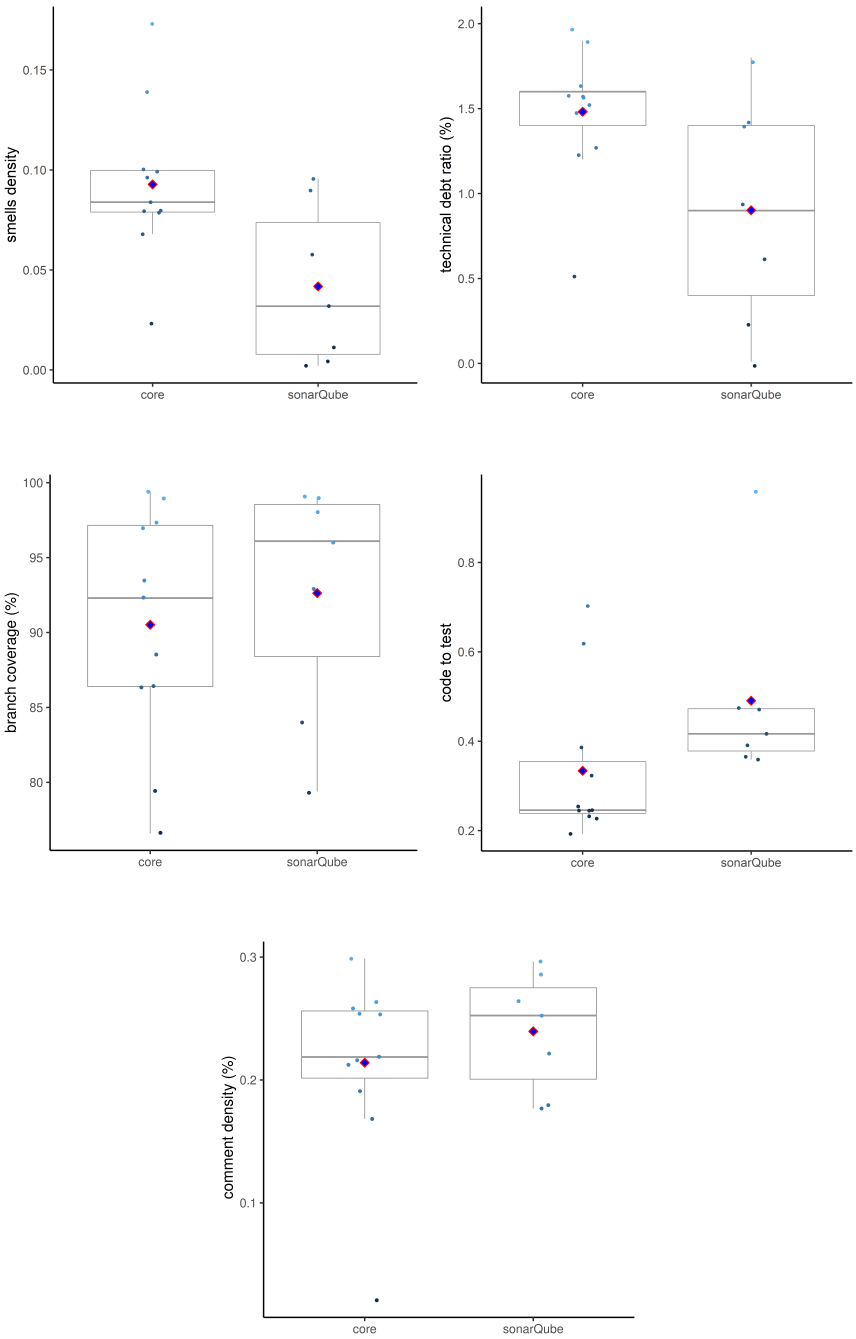

Fig. 3. RQ1. Evolution of the measures for TD indicators with the training treatment (core vs. sonarqube) in the project (programming task) developed in pairs (p3).

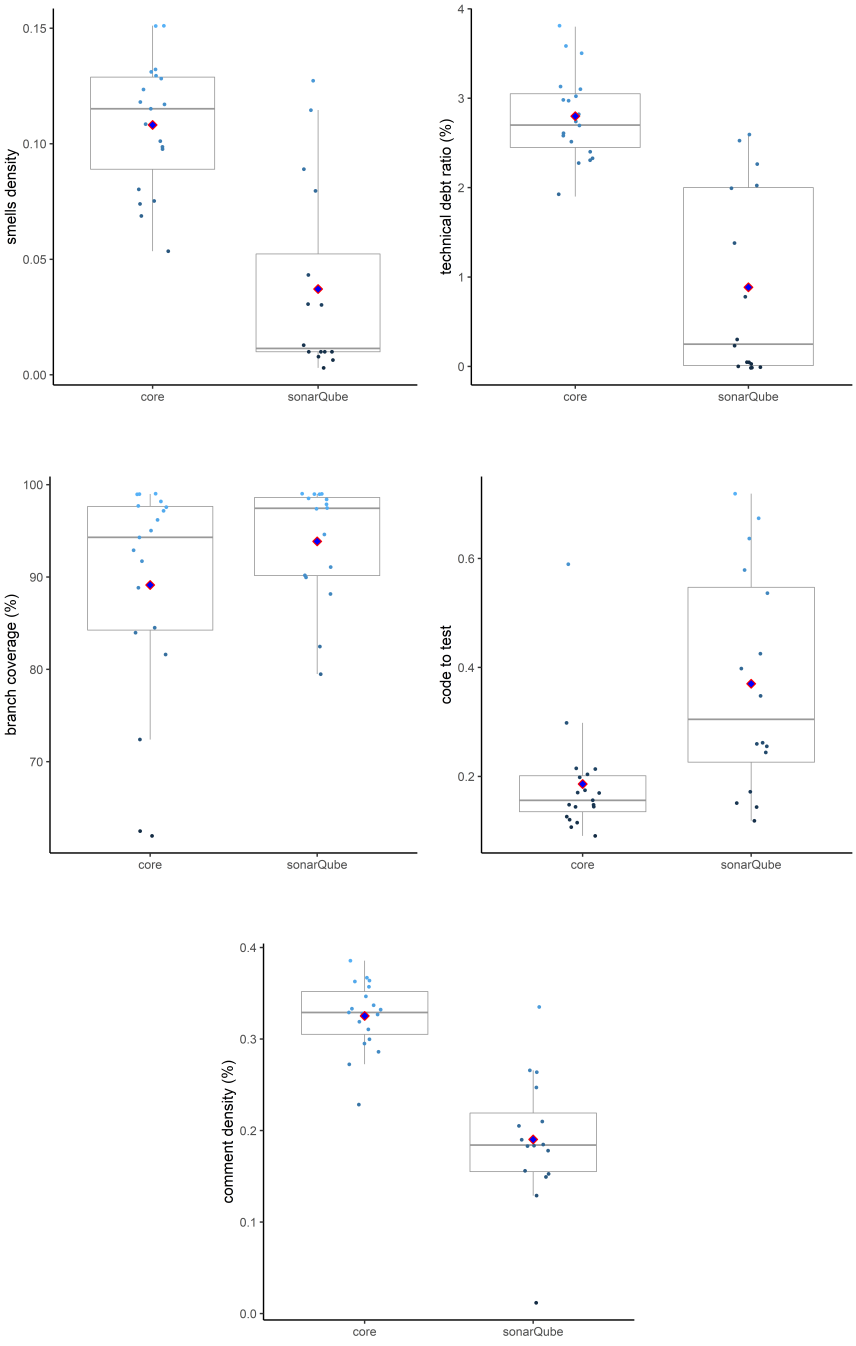

Fig. 4. RQ1. Evolution of the measures for TD indicators with the training treatment (core vs. sonarqube) in the project (programming task) developed individually (p4).

The interpretation of SD and TDR means that values equal to zero are preferred. Taking the mean of the TDR values, a significant improvement is observed at p3 and even more improvement at p4. The analysis of the mean values with respect to SD also shows this improvement, even more so in p4.

The interpretation of the C2T measure is different, where values close to 1 are better in this metric. The results for the improvement of C2T compared to the mean on p3 and p4 are again confirmed.

In both scenarios, p3 and p4, the Branch Coverage (BC) measure is shown to be independent of the training treatment. The interpretation of the BC measure denotes the best results with values close to 1. If we analyse in detail the values of its means, we observe very high values (> 0.9) and there is a slight improvement in its values when training with `SonarQube`.

The only indicator that behaves differently for p3 and p4 is Comment Density (CD). In p3, it behaves the same for both training treatments. In p4, different behaviours are observed. The interpretation of this measure is not objective and is based on recommendations of threshold values that even depend on programming languages. For this reason, it is not possible to analyse the improvement based on averages.

As a conclusion of this analysis, it has been observed that training students by continuously raising TD awareness using `SonarQube` improves the measures related to design/code debt and test debt indicators in their developments. The measure related to documentation debt behaves differently in the case of the p4 project, but we cannot state that it is improved.

Regarding RQ2: *Does introducing gamification in technical debt management improve technical debt indicators?*

For each metric considered, $\forall M_i \in \{SD, TDR, BC, C2T, CD\}$, and for project (programming task) $p4$, we analysed the null hypothesis:

$$H_0^{RQ2}(p4) : \mu_{sonarqube}^{M_i} = \mu_{core}^{M_i}$$

Table 5 collects the results of the Student's t-test on the unique experimental scenario considered (individual project p4, since it is the one in which gamification is applied, as explained in Section 3.5.3). The cells in Table 5 represent the p-values. The p-values denoting different behaviours of the measure applying the different training treatments (sonarqube vs. gamification) are marked in bold, according to the following degree criterion: less than 0.001 (highlighted with '***'), less than 0.01 (highlighted with '**') and less than 0.05 (highlighted with '*').

The experimental results in Table 5 suggest that the training treatment that adds gamification based on a contest and ranking, with respect to continuously raising TD awareness using `SonarQube`, does not have a significant influence on the measures. According to the p-values and the degree criteria, 3 of the 5 measures treated show the same behaviour with regard to the averages of the

Table 5. Quantitative analysis RQ2. p-value Student's t-test and effect size computation with Cohen's d (sonarqube vs. gamification).

	Smells Density (SD)	Technical Debt Ratio (TDR)	Branch Coverage (BC)	Code to Test (C2T)	Comment Density (CD)
P4 Student-t	0.3461	0.6922	0.1201	**0.0005*** **	**0.0001*** **
effect size	-0.39 small	0.04 very small	-0.69 medium	**1.16 large**	**1.42 large**

metrics. These measures correspond to Branch Coverage (BC), Smells density (SD) and Technical Debt Ratio (TDR).

Figure 5 shows the boxplots of each metric comparing the sonarqube (TD awareness training treatment, with respect to the gamification treatment, using the TD as a reference value. The visual analysis corroborates the results of the comparison obtained with the hypothesis tests of Table 5.

The interpretation of the C2T metric is that it improves when the values are closer to 1. The results of C2T denote that adding gamification does not improve the mean (see Fig. 5).

Regarding the CD metric, there is an increase in the mean in the gamification training treatment vs. the sonarqube training treatment (from 19% to 27%). However, if we look at the CD when comparing sonarqube treatment vs core treatment, we can see a decrease in the mean. In fact, it can be interpreted that raising TD awareness using `SonarQube` has not really contributed to the quality of the javadoc comments. When the instructor observed this drop from the core training treatment to the sonarqube training treatment, in the gamification treatment it was highly emphasised in the lessons and seminars. CD did not participate in any way in the game rules, so we believe that the CD is not influenced by the gamification, but by the instructor's reinforcement.

5 Discussion

5.1 RQ1: Raising developers' awareness of Technical Debt

As noted by other authors, after reviewing the results that answer our research question RQ1, we have observed the advantages in the improvement in TD indicators when the tool `SonarQube` is included. Specifically, in our study, we have confirmed this with indicators related to design/code debt, test debt and documentation debt. These results agree with other previous works, such as those described in Sec. 2. It seems evident that continuously raising awareness of TD will have a positive effect on TD indicators. However, as with other similar ones, this intuition should be empirically confirmed, as is done here.

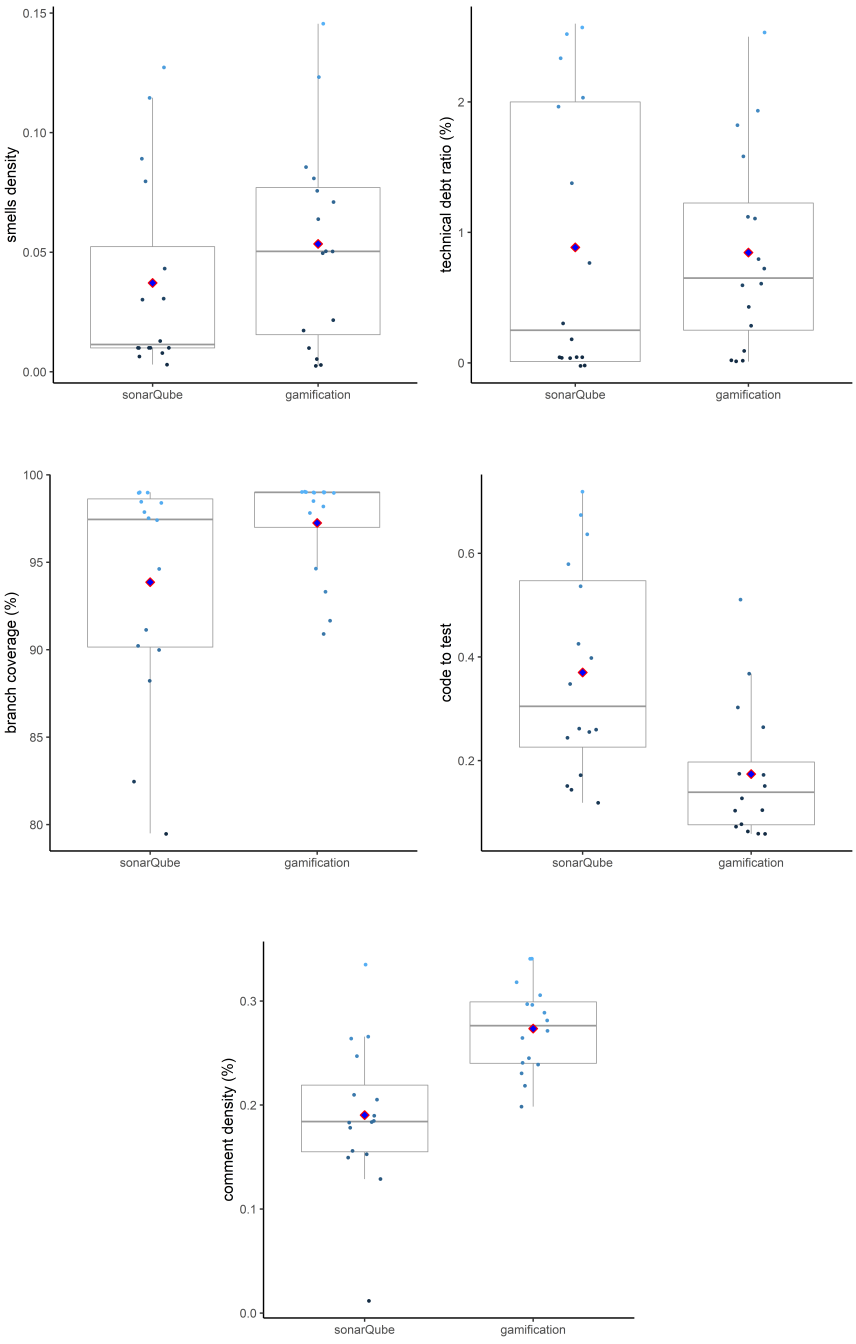

Fig. 5. RQ2. Evolution of the measures for TD indicators with the training treatment (sonarqube vs. gamification) in the programming task developed individually (p4).

From a more detailed point of view, with this study, we observe that the types of TD analysed, such as design/code debt, test debt and documentation debt, obtain different significant results. Documentation debt is the least influenced by the training treatment. This may be due to the interpretation of the indicator used. Comment density is an objective indicator whose interpretation does not lead unambiguously to indicating whether a higher value means better or worse documentation. There is a non-capture subjectivity component to the quality of the documentation.

In the case of BC (branch coverage), the use of `SonarQube` does not add value to the use of JUnit+EclEmma in Eclipse. `SonarQube` just imports the results of the report generated by JaCoCo. That is probably why this test debt indicator is not significantly improved.

Regarding C2T improvement, overviews of the number of tests, classes, of code, functions, etc, are very clearly shown in `SonarQube`. It is easier to calculate the C2T indicator with the information that can be consulted in `SonarQube` than with the aggregated information from the different IDE plugins.

In the case of SD (Smell Density), participants in the core treatment were instructed to use pmd as a plugin in Eclipse, as pmd detects smells in code. `SonarQube` detects a wider set of smells, as it is an aggregation of different plugins, pmd in particular. This may influence the improvement of this indicator in the treatment called sonarqube.

As for TDR (Technical Debt Ratio), the inclusion in the process of the continuous use of `SonarQube` seems to make the difference. Using several plugins to obtain metrics and detect smells, such as Eclipse Metrics and pmd, does not seem to foster raising awareness of TD. We believe that being aware of the increase in TD at several points of the development process and, at the same time, of the assessment in the time needed to repay it, influences participants' attempts to keep TD under control. Furthermore, using these indicators as means of detecting refactoring opportunities and repaying TD in this part of the TDD cycle is another factor to consider in the observed differences.

Most participants added white-box tests to increase the coverage indicator related to test debt. To improve the design and code debt indicators, participants applied refactorings. We have observed that the majority of the participants applied some manual refactorings. We also observed the use of automatic refactoring operations with the Eclipse Refactoring plugin. The most common were "renames" refactorings. Almost all participants used "renames" (methods, attributes), followed by "moves" refactorings (classes from one package to another, methods from one class to another). Next in usage was "Use supertype as possible" followed by "Extract Interface". Last, and least, "Extract method" refactoring was also applied several times by different participants. We did not find other kinds of automatic refactoring applied. In the rest of the cases, when participants need to repay TD, they refactor by manual edition. This also applies in the case of the gamification treatment.

Results in p4 are better than in p3 mainly because they are individually applying what they learned and experienced in pairs. However, we believe it

is also because of the emphasis in the instructions that explicitly mention the actions to perform in the last phase of the TDD cycle. All these are available in the course documentation, but it seems to raise awareness when it is emphasised in the task programming instructions.

5.2 RQ2: Gamification on Technical Debt management

Regarding the observations of adding a competitive factor, based on ranking gamification, to the TD management (RQ2), no improvement in results has been observed, compared to continuously raising awareness of TD using `SonarQube`. Although it seems interesting to continue doing studies to confirm these observations, it has already been pointed out in Moldon et al. [34] that the inclusion of gamification elements can end up conditioning the behaviour of software developers, even in a negative sense. However, it is debatable whether the inclusion of `SonarQube` for raising awareness of TD is implicitly already a gamification activity embedded in the development process. Developers can identify TD measures with points and TD reduction with achievements. Given our results, the inclusion in the second training treatment of the management of TD with a tool, before gamification in the third training treatment, may have altered or softened this effect. This raises the possibility of improving the gamification activity itself.

The participants in this study, the 18-19 group, indicated that they missed the social part of gamification e.g. showing their achievements to the community, increasing competitiveness and having immediate feedback of the impact in the ranking of their actions repaying the debt. Introducing all these requires a more sophisticated gamification tool. After this study, we adapted a competitive programming environment and showed the first results of using it in improving technical debt indicators as reported in [14].

The participants also indicated that when they did not know what else to do, they devoted time to thinking how to programme the same with lower cyclomatic complexity in order to indirectly obtain better C2T and to improve javadoc comments in order to obtain better marks.

Further research is needed and a new study should be designed to check whether different decisions in the gamification (rules and environment) work better for the objective of having better technical debt indicators.

5.3 Implications

As a consequence of the results and their discussion, we can highlight some implications for academia, researchers and practitioners.

Academia has to include TD in their curricula regarding Software Engineering. In such cases, TD should be treated as a "first class subject" as are Requirements or Testing [20]. University teachers need to incorporate TD concepts and use tools (such as `SonarQube`) that allow students to acquire some TD skills. Teachers should also be involved in the use and evaluation of different gamification techniques in Computer Science courses.

It is important for researchers to be involved in studies on the effect of the use of tools on TD awareness and management. They should also investigate to what extent this influences the improvement of TD indicators. It is also necessary, on the one hand, to continue the empirical validation of the effectiveness, efficacy and efficiency of gamification environments; and, on the other, to investigate how gamification theory and tools should be used to create new and more efficient gamification environments.

In the case of practitioners, it is necessary to implant such tools as `SonarQube` in software engineering teams, and continuously assess their impact on decreasing TD or keeping it under control. On the one hand, industry has to incorporate TD into CI/CD processes. On the other, companies should adopt gamification techniques very cautiously and should not take it for granted that they will produce an improvement in software quality, specifically in the management of TD. As is fortunately being accepted more and more in the software industry, the importance of quantitative and data-driven management must be reaffirmed when managing software quality and TD. Finally, it is important to find the best ways to train practitioners in aspects related to TD.

6 Threats to Validity

In this section, we discuss threats to the validity of our results using Cook and Campbell's classification [4, 13, 45, 55]. The threats are classified as conclusion, internal, construct and external validity.

6.1 Conclusion validity

Threats to the conclusion validity refer to the reliability of the conclusions, it describes the relationship between the treatment and the outcome of the experiment [55]. One of the threats could be the reliability of the experiment's implementation. In order to avoid this threat it has been implemented in a similar way in each training treatment. Furthermore, the type of problem assigned as projects was conceptually similar, but also similar in size and complexity.

As Ampatzoglou et al. [4] state, researcher bias can greatly impact the conclusions reached and can be considered a threat to the conclusion validity. In this case, the interpretation of the comment density metric is subjective and its improvement is evaluated according to threshold values that are highly dependent on such factors as the programming language or the organisation's commenting style (e.g., inclusion of code licensing information in the comments).

In our study, the comparison of treatment methods has been one of equality and difference, so this threat has had no effect.

6.2 Internal validity

Threats to internal validity refer to ensuring that anything that may affect the dependent variables is a causal relationship and not the result of a factor that has not been accounted for or measured.

We mitigated the threat of subject dropout in the middle of the study by making it count as part of the course grade. Nevertheless, it is impossible to fully avoid dropouts. Those objects resulting from dropouts were removed from the study because they could not be analysed.

We also reduced the threat of history, as each training treatment was carried out during the same academic semester in successive courses, in which similar events occur.

The aim is to strike a balance between avoiding the copying effect from one course to another using the same programming task and not comparing the treatment results from one year to another using the same programming task. To achieve an adequate compromise, it was decided to avoid the copy effect but to introduce programming tasks, which, although different, are very similar in terms of size, complexity, and hours of effort required. This intention was checked with an *a posteriori* study shown previously in Section 3.7.1. This threat could be eliminated in a replication in which the experiment can be carried out with 3 groups in parallel and with no communication between them.

The group subjects are different in each training treatment, so the results may be biased by the knowledge they have at the start of training in each training treatment. Given the context in which the quasi-experiment design could be conducted, it was not possible to control the subjects in each of the training treatments. However, in order to reduce this threat, it was possible to control their educational background, ensuring that the subjects had no previous work experience and generally had the same prior education. However, it is true that a participant may have had some prior knowledge of his/her own.

A threat has been detected in the design of the gamification activity developed in the third training treatment. The gamification definition is based on metrics obtained with `SonarQube`, so it becomes difficult to separate the influence of gamification per se from the influence of the awareness of TD itself. This should be mitigated in future redesigns of the quasi-experiment.

However, from the point of view of a social threat, the effect of gamification in other contexts has not always had positive effects, either from the point of view of excessive disclosure of private information [52] or suspicion of its use as a control mechanism [28]. Although the first problem is not so applicable in gamification in software development; in the second case, it must be taken into account, but more from the point of view of TD management.

6.3 Construct validity

Construct validity refers to the relationship between theory and observation. It is therefore necessary to take into account, on the one hand, the way in which the experiment has been designed and its capacity to reflect the construct, and on the other, those related to the behaviour of the subjects [55].

In our case, the objective is to measure the impact on TD, specifically on four types of TD. In the quasi-experiment, commonly accepted metrics are used in relation to these aspects, so this threat is not important.

As for the objects of study, the projects vary for each training treatment to avoid the learning-copying effect. In this case, the threat was mitigated because the degree of difficulty was similar, which is corroborated by the measures of complexity and size of the projects.

One threat to take into account is the gamification design itself. A series of decisions were taken to design both the gamification process and specifically the contest. It is necessary to point out that it is not possible to mitigate this threat, because the contest is established in that way, but the replication package can always be used to perform another type of gamification.

6.4 External validity

Threats to external validity limit the generalisability of the results, so the subjects, the environment and the timing of the quasi-experiment must all be taken into account.

One of the threats to external validity comes from having students as subjects. In this case, this threat is reduced since they are third-year students who are very close to graduating and starting their professional work.

There is an open debate about the use of novice programmers for this type of studies [21]. Some works have shown that years of professional experience do not have the expected effect on improving quality and productivity. However, experience gained in academic training does seem to have a positive effect [18]. Furthermore, it has also been found that the effect of experience for the introduction of TD is not as relevant as other factors, such as the maturity of the project, the habits of the programmer, or the type of specific tasks developed in the period [19].

The effect of gender on gamification must also be taken into account. Some work has reflected a gender moderation between playfulness and intention to use [12]. In the world of software development, the gender distribution is very uneven, with a higher number of men, more likely to actively participate in these dynamics. In our study, there is hardly any female representation,which is unfortunately representative of what is happening in Software Engineering today. Being aware of this, the study still needs to be replicated in a context where there is more female representation.

Another threat corresponds to the choice of the programming tasks to be solved by the subjects. They are similar to real situations that are developed in industry, limited in complexity, length and duration, but not really a part of a real software project development.

The use of `SonarQube` and its associated plugins as reference tools can also generate some threats. In [44], the Sonar TD plugin is presented as the most cited tool in TD management. This is confirmed by data on 100K organisations, including 15K public open source projects using the tool [46]. Despite being the most widely used, it is true that the tool cannot be 100% correct, with diffuse TD elements and overestimated remediation time [7, 46]. Nevertheless, being one of the de facto standards, it seems to be a suitable option to establish comparisons between experimental studies, while still being aware of the tool's weaknesses.

However, it is considered an external threat to validity, as a generalization threat, because different results could be obtained if another tool were used to measure TD indicators, so it would be interesting to replicate the study by adapting it to other tools. For example, using Kiuwan [8] or Cast Software [9], but given their different characteristics, they would not guarantee the same results. Besides, the high costs associated with this kind of tool make it problematic to replicate them in academic contexts.

In short, the study has been carried out in a specific context. Factors influencing this context may affect the generalisability of the results. It would be interesting to design replications of this study in different contexts and with other programming languages and tools.

7 Conclusions and future work

The importance of managing TD is widely accepted in the software engineering community in both business and academic domains. However, previous training of developers and TD management are different in software development organisations. One reason is theoretical, due to the lack of a concrete definition of the factors for calculating TD. The second reason is practical, and refers to the lack of a standardised technological development environment based on rules and code quality measures.

The literature reviews on TD [31, 44] include studies to obtain a classification and improve the understanding of this metaphorical concept. Under this referential and theoretical framework, it seems relevant to continue developing empirical studies that help to confirm hypotheses related to the consequences of the different ways of managing TD.

In this paper, we have conducted a quasi-experiment based on three training treatments, where we analyse, on the one hand, the effect of continuously raising awareness of TD using such dedicated tools as `SonarQube` and, on the other, introducing gamification in the management of TD, using Software Engineering developers/students as the study subjects.

The study has evaluated the quality improvement with code metrics as indicators of different types of TD (design debt, code debt, test debt and documentation debt). The introduction of `SonarQube` in the development process for continuously raising awareness of TD has significantly improved the studied indicators.

Regarding design/code debt, the results show a significant improvement in this case. However, there is not such a positive effect with the introduction of gamification on TD management.

Regarding the test debt related indicators, it was observed that although the test coverage (BC) did not show an improvement with the introduction of `SonarQube` in the development process, the C2T, which measures the number of unit tests with respect to the production lines of code, did improve. Analysing these

[8] https://www.kiuwan.com
[9] https://www.castsofware.com

results in detail, we observed that, in the baseline technological environment (core treatment), EclEmma was used to obtain reports on the test coverage. In all the treatments, the developments showed a coverage value above 90%. Being aware of this information and setting a quality standard based on a threshold value makes developers comply with it in their developments. However, this test coverage indicator is not conclusive on test debt. In order to see the quality of the test codes, it should be complemented with the C2T, which did improve when `SonarQube` was introduced to continuously raise awareness of TD.

The results for the documentation debt indicators did not show convergent findings.

As a general conclusion of the results obtained in this empirical study, it has been observed that the quality of the codes produced by developers who use tools (such as `SonarQube`) that calculate, and can be used in a systematic way to raise awareness on TD, are better than those produced by developers who do not. Accordingly, investment in staff training in this kind of tool and its proper introduction in the development process do return improvements in TD indicators.

TD indicators, when introducing gamification in TD management, did not reveal a significant improvement compared to when `SonarQube` was introduced to continuously raise awareness of TD. However, it would be advisable to define new empirical studies that consider different gamification strategies (badges, live ranking and social issues) that could achieve an improvement in the TD indicators. These new gamification strategies should take into account, as pointed out in Moldon et al. [33], that the effect of gamification, or in this specific case, the abandoning of certain gamification practices in development, shows a change in the behaviour of programmers; which is not always aimed at improving the quality of the product, but due to other social and psychological issues. Directing gamification under the umbrella of improving code quality and reducing TD should limit and cushion this possible negative effect.

As future work, the most important goal to achieve would be to adapt the study to an industrial setting, and also try to move from a quasi-experiment to experiment, attempting to achieve the full randomisation of the groups. However, proposing new gamification strategies and carrying out experiments to check which strategy brings better results would be another important step forward.

Acknowledgement

This research is based on the work carried out under the Innovation projects of Universidad de Valladolid PID2017/2018-28, PID2018/2019-38. The authors would like to thank the reviewers and Dr. Valentín Cardeñoso for the comments that have served to improve the first versions of this work.

References

1. Alhammad, M.M., Moreno, A.M.: Gamification in software engineering education: A systematic mapping. JOURNAL OF SYSTEMS AND SOFTWARE **141**, 131–

150 (JUL 2018). https://doi.org/10.1016/j.jss.2018.03.065
2. Alhammad, M.M., Moreno, A.M.: Challenges of gamification in software process improvement. Journal of Software: Evolution and Process **32**(6), e2231 (2020). https://doi.org/https://doi.org/10.1002/smr.2231, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2231, e2231 JSME-19-0049.R1
3. Allman, E.: Managing technical debt. Communications of ACM **55**(5) (May 2012)
4. Ampatzoglou, A., Bibi, S., Avgeriou, P., Verbeek, M., Chatzigeorgiou, A.: Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Information and Software Technology **106**, 201–230 (2019). https://doi.org/https://doi.org/10.1016/j.infsof.2018.10.006, https://www.sciencedirect.com/science/article/pii/S0950584918302106
5. Atal, R., Sureka, A.: Anukarna: A software engineering simulation game for teaching practical decision making in peer code review. In: Lichter, H., Anwar, T., Sunetnanta, T., Vianden, M., Dubey, A., Celis, L.E., Grant, E.S., Shankararaman, V. (eds.) QuASoQ/WAWSE/CMCE@APSEC. CEUR Workshop Proceedings, vol. 1519, pp. 63–70. CEUR-WS.org (2015), http://dblp.uni-trier.de/db/conf/apsec/quasoq2015.html#AtalS15
6. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports **6**(4), 110–138 (2016). https://doi.org/10.4230/DagRep.6.4.110, http://drops.dagstuhl.de/opus/volltexte/2016/6693
7. Baldassarre, M.T., Lenarduzzi, V., Romano, S., Saarimäki, N.: On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. Information and Software Technology **128**, 106377 (2020). https://doi.org/https://doi.org/10.1016/j.infsof.2020.106377, https://www.sciencedirect.com/science/article/pii/S0950584919302113
8. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering **SE-10**(6), 728–738 (1984). https://doi.org/10.1109/TSE.1984.5010301
9. Ben-Shachar, M.S., Lüdecke, D., Makowski, D.: effectsize: Estimation of effect size indices and standardized parameters. Journal of Open Source Software **5**(56), 2815 (2020). https://doi.org/10.21105/joss.02815, https://doi.org/10.21105/joss.02815
10. Besker, T., Martini, A., Bosch, J.: Carrot and stick approaches when managing technical debt. In: Proceedings of the 3rd International Conference on Technical Debt. p. 21–30. TechDebt '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3387906.3388619, https://doi.org/10.1145/3387906.3388619
11. Besker, T., Martini, A., Bosch, J.: The use of incentives to promote technical debt management. Information and Software Technology **142**, 106740 (2022). https://doi.org/https://doi.org/10.1016/j.infsof.2021.106740, https://www.sciencedirect.com/science/article/pii/S0950584921001907
12. Codish, D., Ravid, G.: Gender moderation in gamification: Does one size fit all? In: Proceedings of the 50th Hawaii international conference on system sciences. p. 10 (01 2017). https://doi.org/10.24251/HICSS.2017.244
13. Cook, T., Campbell, D.: Quasi-experimentation – design and analysis issues for field settings. Houghton Mifflin Company (1979)
14. Crespo, Y., Gonzalez-Escribano, A., Piattini, M.: Carrot and stick approaches revisited when managing technical debt in an educational context. In: 2021 2021 IEEE/ACM International Conference on Technical Debt (TechDebt) (TechDebt). pp. 99–108. IEEE Computer Society, Los Alamitos,

CA, USA (may 2021). https://doi.org/10.1109/TechDebt52882.2021.00020, https://doi.ieeecomputersociety.org/10.1109/TechDebt52882.2021.00020
15. Cribari-Neto, F., Zeileis, A.: Beta regression in R. Journal of Statistical Software **34**(2), 1–24 (2010), http://www.jstatsoft.org/v34/i02/
16. Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger **4**(2), 29–30 (1993). https://doi.org/10.1145/157710.157715
17. Delgado, D., Velasco, A., Aponte, J., Marcus, A.: Evolving a project-based software engineering course: a case study. In: The 30th IEEE Conference on Software Engineering Education and Training. pp. 77–86 (2017)
18. Dieste, O., Aranda, A.M., Uyaguari, F., Turhan, B., Tosun, A., Fucci, D., Oivo, M., Juristo, N.: Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. Empirical Software Engineering **22**(5), 2457–2542 (Oct 2017). https://doi.org/10.1007/s10664-016-9471-3, https://doi.org/10.1007/s10664-016-9471-3
19. Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: The temporality of technical debt introduction on new code and confounding factors. Software Quality Journal (11 2021). https://doi.org/10.1007/s11219-021-09569-8
20. Falessi, D., Kruchten, P.: Five reasons for including technical debt in the software engineering curriculum. In: Proceedings of the 2015 European Conference on Software Architecture Workshops. ECSAW '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2797433.2797462, https://doi.org/10.1145/2797433.2797462
21. Feldt, R., Zimmermann, T., Bergersen, G.R., Falessi, D., Jedlitschka, A., Juristo, N., Münch, J., Oivo, M., Runeson, P., Shepperd, M., Sjøberg, D.I.K., Turhan, B.: Four commentaries on the use of students and professionals in empirical software engineering experiments. Empirical Software Engineering **23**(6), 3801–3820 (Dec 2018). https://doi.org/10.1007/s10664-018-9655-0, https://doi.org/10.1007/s10664-018-9655-0
22. Ferrari, S., Cribari-Neto, F.: Beta regression for modelling rates and proportions. Journal of Applied Statistics **31**(7), 799–815 (2004), https://EconPapers.repec.org/RePEc:taf:japsta:v:31:y:2004:i:7:p:799-815
23. Foucault, M., Blanc, X., Storey, M.D., Falleri, J., Teyton, C.: Gamification: a game changer for managing technical debt? A design study. CoRR **abs/1802.02693** (2018), http://arxiv.org/abs/1802.02693
24. Fowler, M., with contributions of Beck, K., Brant, J., Opdyke, W., et al.: Refactoring: Improving the Design of Existing Code. Object technology series, Addison-Wesley (1999)
25. Fowler, M.: Technical debt quadrant. MartinFowler.com blog (2009)
26. Fox, A., Patterson, D.: Engineering Software as a Service: An Agile Approach Using Cloud Computing. Strawberry Canyon LLC, 2nd edn. (2013)
27. Haendler, T., Neumann, G.: Serious refactoring games. In: Proceedings of the 52nd Hawaii International Conference on System Sciences. pp. 1–10 (2019). https://doi.org/10.24251/HICSS.2019.927
28. Hammedi, W., Leclercq, T., Poncin, I., Alkire (Née Nasr), L.: Uncovering the dark side of gamification at work: Impacts on engagement and well-being. Journal of Business Research **122**, 256–269 (2021). https://doi.org/https://doi.org/10.1016/j.jbusres.2020.08.032, https://www.sciencedirect.com/science/article/pii/S0148296320305415
29. Kitchenham, B.A., Pretorius, R., Budgen, D., Brereton, P., Turner, M., Niazi, M., Linkman, S.G.: Systematic literature reviews in software engineering - A tertiary study. Inf. Softw. Technol. **52**(8), 792–805 (2010).

https://doi.org/10.1016/j.infsof.2010.03.006, https://doi.org/10.1016/j.infsof.2010.03.006
30. Lenth, R.: emmeans: Estimated Marginal Means, aka Least-Squares Means (2020), https://CRAN.R-project.org/package=emmeans, r package version 1.5.2-1 — For new features, see the 'Changelog' file (in the package source)
31. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. JOURNAL OF SYSTEMS AND SOFTWARE **101**, 193–220 (MAR 2015). https://doi.org/10.1016/j.jss.2014.12.027
32. Liu, X., Woo, G.: Applying code quality detection in online programming judge. In: Proceedings of the 2020 5th International Conference on Intelligent Information Technology. p. 56–60. ICIIT 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385209.3385226, https://doi.org/10.1145/3385209.3385226
33. Moldon, L., Strohmaier, M., Wachs, J.: How gamification affects software developers: Cautionary evidence from a natural experiment on github. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 549–561. IEEE Computer Society, Los Alamitos, CA, USA (may 2021). https://doi.org/10.1109/ICSE43902.2021.00058, https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00058
34. Moldon, L., Strohmaier, M., Wachs, J.: How gamification affects software developers: Cautionary evidence from a quasi-experiment on github. ArXiv **abs/2006.02371** (06 2020)
35. Monteiro, R.H.B., de Almeida Souza, M.R., Oliveira, S.R.B., dos Santos Portela, C., de Cristo Lobato, C.E.: The diversity of gamification evaluation in the software engineering education and industry: Trends, comparisons and gaps. CoRR **abs/2102.05089** (2021), https://arxiv.org/abs/2102.05089
36. Nguyen Quang Do, L., Bodden, E.: Gamifying static analysis. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 714–718. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3264830, https://doi.org/10.1145/3236024.3264830
37. Paolino, P.: Maximum likelihood estimation of models with beta-distributed dependent variables. Political Analysis **9**(4), 325–346 (2001)
38. Parodi, E., Matalonga, S., Macchi, D., Solari, M.: Comparing technical debt in student exercises using test driven development, test last and ad hoc programming. In: 2016 XLII Latin American Computing Conference (CLEI). pp. 1–10 (2016). https://doi.org/10.1109/CLEI.2016.7833380
39. de Paula Porto, D., de Jesus, G.M., Ferrari, F.C., Fabbri, S.C.P.F.: Initiatives and challenges of using gamification in software engineering: A systematic mapping. Journal of Systems and Software **173**, 110870 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2020.110870, http://www.sciencedirect.com/science/article/pii/S0164121220302600
40. Pedreira, O., Garcia, F., Brisaboa, N., Piattini, M.: Gamification in software engineering - A systematic mapping. INFORMATION AND SOFTWARE TECHNOLOGY **57**, 157–168 (JAN 2015). https://doi.org/10.1016/j.infsof.2014.08.007
41. Quezada Sarmiento, P., Guaman, D., Barba Guamán, L.R., Enciso, L., Cabrera, P.: Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis. In: Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering. pp. 171–175 (07 2017)

42. Raibulet, C., Arcelli Fontana, F.: Collaborative and teamwork software development in an undergraduate software engineering course. The Journal of Systems & Software **144**, 409 – 422 (2018)
43. Ramasubbu, N., Kemerer, C.F.: Integrating technical debt management and software quality management processes: A normative framework and field tests. IEEE Transactions on Software Engineering **45**(3), 285–300 (2019). https://doi.org/10.1109/TSE.2017.2774832
44. Rios, N., de Mendonça Neto, M.G., Spínola, R.O.: A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology **102**, 117 – 145 (2018). https://doi.org/https://doi.org/10.1016/j.infsof.2018.05.010, http://www.sciencedirect.com/science/article/pii/S0950584918300946
45. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131 (Dec 2008). https://doi.org/10.1007/s10664-008-9102-8, https://doi.org/10.1007/s10664-008-9102-8
46. Saarimaki, N., Baldassarre, M.T., Lenarduzzi, V., Romano, S.: On the accuracy of sonarqube technical debt remediation time. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 317–324 (2019). https://doi.org/10.1109/SEAA.2019.00055
47. Silva, D., Nunes, I., Terra, R.: Investigating code quality tools in the context of software engineering education. Computer Applications in Engineering Education **25**(2), 230–241 (2017). https://doi.org/10.1002/cae.21793, https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.21793
48. Souza, M.R.d.A., Veado, L., Moreira, R.T., Figueiredo, E., Costa, H.: A systematic mapping study on game-related methods for software engineering education. INFORMATION AND SOFTWARE TECHNOLOGY **95**, 201–218 (MAR 2018). https://doi.org/10.1016/j.infsof.2017.09.014
49. Stepsize: The State of Technical Debt 2021 report. https://www.stepsize.com/report (2021), [Online; accessed 14-July-2021]
50. Stol, K.J., Schaarschmidt, M., Goldblit, S.: Gamification in software engineering: The mediating role of developer engagement and job satisfaction. Empirical Software Engineering (2021)
51. Tonin, G.S., Goldman, A., Seaman, C., Pina, D.: Effects of technical debt awareness: A classroom study. In: Baumeister, H., Lichter, H., Riebisch, M. (eds.) Agile Processes in Software Engineering and Extreme Programming. pp. 84–100. Springer International Publishing, Cham (2017)
52. Trang, S., Weiger, W.H.: The perils of gamification: Does engaging with gamified services increase users' willingness to disclose personal information? Computers in Human Behavior **116**, 106644 (2021). https://doi.org/https://doi.org/10.1016/j.chb.2020.106644, https://www.sciencedirect.com/science/article/pii/S0747563220303915
53. Wickham, H.: ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York (2016), https://ggplot2.tidyverse.org
54. Wickham, H.: readr: Read Rectangular Text Data (2020), https://CRAN.R-project.org/package=readr, r package version 1.5.2-1 — For new features, see the 'Changelog' file (in the package source)
55. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 2012. edn. (2012)

A Programming tasks assigned

A.1 Description of each programming task

A.1.1 Description of the programming task: P3-core treatment

In an information system, it is necessary to develop the services of a newsletter. The following is a description of the features to be included in the current iteration:

Concerning the news:

1. A news item necessarily has a headline, the date of publication, the source (name of the media or press agency), a URL where the complete content of the news item is found, and a category.
2. The categories in which a news item can be classified are: national, international, society, economy, sport, and culture.
3. News headlines, according to good practices in this regard, should have less than 13 words and at least one. One-word headlines are rare but acceptable, for example: "Shame!".
4. A news item should be comparable to another item in terms of date of publication, indicating whether it is earlier, the same, or later.
5. A news item will be considered "similar" to another if it coincides in the headline, category, and date of publication, even if the source and URL with the complete content do not coincide. In the case of the date of publication, it will also be considered similar with two days of difference in the date (previous or later).

A newsletter is a grouping and selection of news for some purpose. Concerning the newsletter:

1. It must be possible to create an empty newsletter.
2. It must be possible to add news to a newsletter.
3. The newsletter must ensure that it does not contain repeated news items.
4. It must be possible to create a newsletter from a list of news items. The order of appearance in the list shall be considered the order of arrival of the news items in the newsletter.
5. It must be possible to know how many news items are in a newsletter.
6. For each newsletter it must be possible to know the date of the most recent news contained in the newsletter and the date of the oldest.
7. It must be possible to obtain from the newsletter a list in chronological order (from earlier to later) based on the date of publication of the news. When two news items coincide in date of publication, the one that was first added to the newsletter shall be considered the earlier one.
8. It must be possible to obtain from the newsletter a list of its news ordered by category. This list will always appear in the order in which they are mentioned in point 2 of the description of the news items. That is to say, first the national news, then the international news, etc. Within each category, the news will appear in chronological order applying the same criteria as in point 7.

9. A list of news items "similar" to a given news item contained in the newsletter can be obtained from the newsletter, sorted according to the same criteria as in point 7.
10. From a newsletter, another newsletter may be obtained whose news will be a subset of those contained in the origin. Several possibilities will be considered:
    (a) It must be possible to obtain a newsletter from another given a specific date, it is ensured that in the resulting newsletter all the news has been published on the day of the indicated date.
    (b) If you wish to obtain a newsletter from another newsletter given two dates, it is ensured that in the resulting newsletter all the news items have been published within the given date range.
    (c) You can also define one newsletter from another given a category of news. In the resulting newsletter, all the news will be of the given category.
    (d) It must be possible to combine options (a) or (b) with (c), i.e., to be able to obtain a newsletter from another newsletter that includes all the news of a category on a date, or all the news of a category within a date range.

A.1.2 Description of the programming task: P3-sonarqube treatment

A system needs to model the information and behaviour of a city's urban bus network. The features to be included in the current iteration are described below.

Concerning the bus network:

1. The bus network consists of several lines, at least two.
2. A line is uniquely identified in the bus network by a number.
3. It must be possible to query the bus network, given a number, for the line object corresponding to that number.
4. It must be possible to add or remove lines from the network.
5. It must be possible to obtain an array with all the lines that form the network.

Concerning the lines and their stops:

1. Every line has two special stops known as the start and end of the route. The lines have an almost circular route, so the start and end of the route must be very close, at a distance of fewer than 100 metres.
2. It must be possible to obtain an array with all the stops of a line from the start to the end of the route. A line must have at least 3 stops (counting the start and end of the line).
3. It must be possible to add or remove intermediate stops to the lines, as well as to change the start and end of the route (provided that the correction conditions are still met).

4. All stops will be uniquely identified by an order number on the route (starting from the start of the route to the end of the route). Stops will also have a GPS address expressed in decimal degrees (GD).
5. GPS coordinates locate an object's position on the entire Earth. The location is done by characterising the latitude and longitude of the position. Decimal degrees represent the minutes and seconds part of the degrees represented in the sexagesimal system as the decimal part of a real number whose integer part is given by degrees ($2°15'23'' = 12° + 15(1/60)° + 23(1/3600)° \approx 12.25639°$). In decimal degrees (GD) Northern latitudes are positive, Southern latitudes are negative, Eastern longitudes are positive and Western longitudes are negative.
6. In the bus network, given a direction expressed in GD and a radius expressed in metres, information on the lines that stop within that radius should be provided.
7. It must be possible to check if a line has a stop near a GPS address, defining near at a distance of fewer than 200 metres.
8. It must be possible to know if a line has a correspondence with another line. Correspondence between two lines is defined as the case in which a line has a stop near a stop of the other line, defining near as less than 200 metres.
9. It must be possible to know, in case of correspondence, at which stop or stops this correspondence occurs.
10. It must be possible to know if there is the possibility of direct transfer from one line to another . The possibility of direct transfer is defined as both lines having stops that coincide exactly in the GPS position expressed in GD.
11. It must be possible to know, in case there is a possibility of direct transfer, at which stop or stops there is such a possibility.
12. It must be possible to know the distance from one stop to another on the line. We also want to know the distance from a stop on one line to a stop on another line of the network.

A.1.3 Description of the programming task: P3-gamification treatment

It is desirable to have support for a voting system that allows rankings of different elements of the same type to be created. We do not wish to define the whole system at the moment, nor how the end-user will use it.

In the voting system, there will be contests from the results of which rankings will be obtained.

Concerning the contest:

1. It must be possible to open a contest, the result of which will be a ranking. The ranking can be a top 10 or you could choose the number of elements to be included in the ranking.
2. The elements are nominated to participate in the contest. It is desirable to be able to nominate elements for the contest. It is not possible to nominate elements that have already been nominated.

3. It must be possible to nominate several elements at once.
4. It must be possible to close the nominations to start voting.
5. The elements are voted once they have been nominated. It must be possible to vote for one item in the contest. You cannot vote more than once in a contest.
6. To vote a nominated item, an identifier representing the voter must be indicated, but anonymity must be guaranteed (a task that is beyond the scope of this practice).
7. As long as the voting is not closed, a resulting ranking cannot be produced. It must be possible to close the voting in the contest.
8. If the voting has already been closed, it is not possible to nominate more elements or to vote for the already nominated ones, but it is possible to consult the result by obtaining the resulting ranking. It must be possible to obtain the ranking with the result of the voting.
9. In order not to dilute the voting too much, a limit on the number of nominations of elements will be established. It must be possible to set the limit when the contest is created.

Once the resulting ranking has been produced, it must be possible to consult it. Concerning the ranking:

1. It must be possible to consult the element that occupies position number i in the ranking.
2. It must be possible to query the position of a given element e, by value or by reference.
3. It must be possible to know if element e is in the ranking, by value or by reference.
4. It must be ensured that the ranking cannot be modified.
5. It must be possible to compare the ranking with another ranking (of the same size) of elements of the same type and obtain a list with the differences of positions, taking the first ranking as reference (example: [+2, -1, -1, 0] indicates that the first element has moved up 2 positions, the second has moved down 1 position, the third the same, and the fourth and last has not moved from its previous position).

It is also desirable to have a method in the class that implements a contest which compares two already closed contests, receiving the number of elements to compare and the other contest (of elements of the same type as the receiving contest) with which it will be compared as parameters. The implementation of this method will be based on the rankings. It will return the array of differences between the rankings of both contests in comparison (rankings of the size indicated as a parameter).

A.1.4 Description of the programming task: P4-core treatment

We are developing an application for an educational environment. In this environment, we have been asked to provide the following necessary characteristics with respect to the evaluation of a subject.

It must be taken into account that we want the student to be univocally identified by a string of characters that represents a code in another system. The data of the students and their grades should not be in the same system in order to be able to separate them and to better guarantee data protection.

A subject must have a name, a short description, a maximum grade, a start date and an end date.

Tests can be created linked to the subject and they shall have a test date. The completion date of a test in a subject must be included between the start and end dates of the subject.

Each test has a name, a brief description, a maximum grade and a weight in the evaluation of the corresponding period. The weight will be a number between 0 and 1, representing the percentage in which the grade will be considered in the subject.

In the subject, the sum of the weights of the tests cannot be greater than 1. It will not be necessary to always be 1 in order to allow the addition of new tests. Only the sum shall be required to be one at the time of obtaining the final grades for the course, as explained below.

A grade for a test is a pair <student identifier, grade earned on the test>. The grade cannot be higher than the maximum grade indicated for the test. There are two ways to add grades to a test: add a grade (a pair is added) or add a list of grades (of student-grade pairs).

Tests have an indicator as to whether they have been fully graded or not. In a fully graded test, no new grades can be added, but grades already included can be modified.

For a test, you can request the list of grades obtained (list of student-grade pairs). If the date of the test has not been held, it cannot be graded.

For a subject, the list of final grades or the list of partial grades of the subject can be requested (list of student-grade pairs). The grades obtained by the students in the subject may not be higher than the maximum grade assigned to it. It will be taken into account that each test has a maximum grade associated with it, which may differ from each other and from the course grade. The grades of the tests must be weighted in the subject according to the weight.

The final grade list for the course may only be requested if the sum of the weights of the tests is exactly 1 and if all tests have been fully graded.

Partial grades may be obtained at any time, using only the grades of the tests that have been fully graded so far for their calculation.

A.1.5 Description of the programming task: P4-sonarqube treatment

As part of a bigger system, we wish to have a special type of Queue that we will call "Buddy Queue". Normally, you have to enter a queue at the end. A Buddy Queue is where you can reserve a place for a friend or group of friends. Thus, when a new person arrives in the queue, if a friend of his/her has reserved a place, he/she can stand right in front of his/her friend, instead of having to go to the back of the queue.

To avoid situations that are difficult for the rest of the people in the queue to accept, such as the whole group of classmates in Software Engineering being able to sneak in because one reserved places for 40, the following limitations are established:

- It will not be acceptable to enter the queue by reserving for more than 10 friends.
- If a person has reserved for n friends, after the n indicated friends have entered, he/she will not be able to allow anyone else to enter.
- A friend who has been allowed to enter by another friend cannot in turn allow anyone else to enter.
- A person who is in the queue cannot re-enter the queue.

As a minimum, it must be possible to have certain functionalities in the buddy queue that allow:

1. To reserve places in the queue: when a person arrives in the queue, he/she provides the number of friends he/she wishes to reserve for. He/She may not reserve for anyone else. The one who makes the reservation is placed last in the queue.
2. Queuing with the one in the queue: when a new person arrives in the queue, he/she checks if he/she has a friend already in the queue who has booked. If so, he/she can enter the queue just in front of his/her friend (always taking care of the restrictions above), while also making sure that whoever is in the queue wishes him/her to be there as a friend.
3. Knowing how many friends a person said he/she was coming with.
4. Knowing, given a person who is in the queue, how many friends he/she can still sneak in.
5. Knowing, for a person who is already in the queue, which of those in front are the friends for whom he/she reserved a place.
6. Knowing which person would be served according to the order of the queue.
7. Serving the person whose turn it is, after which it would simply be that he/she is no longer in the queue.

In the case of a person, it must be possible to:

1. Know if another person is his/her friend, as well as to know all his/her friends.
2. Have the person meet other people and become friends with them.

3. Differentiate between acquaintances and friends. First, someone must be an acquaintance to then be a friend.
4. Allow the person to stop being friend with one of your present friends, who would again become simply an acquaintance.

A.1.6 Description of the programming task: P4-gamification treatment

The same programming task as in P4-core.

A.2 Instructions given to participants

A.2.1 Instructions given for P3-core treatment

P2 subsumed in P3:

Steps 1 and 2 of the TDD cycle (Red-Green-Refactor)[10] corresponding to the Red phase will be performed:

1. Write the tests that exercise the code you wish to have,
2. Check that the tests fail.

Following the test-driven design (TDD) process:

- define which classes we are going to create
- define the test classes to perform the test-driven design process
- define (in the test classes) how the objects of the classes we have created are used.
- specify their functionality in the tests.
- describe this functionality in the javadoc of the created classes.

The history of *commits* should allow the TDD process to be appreciated.

Once the first version of the tests that allowed us to create the *stubs* of the classes has been established, we should specify its functionality in these tests and describe it in the javadoc. **Improve the tests**. Add new tests (in separate test classes), taking into account black-box techniques (data-driven partition tests, state-based partition tests).

Tests will be implemented with JUnit 4. The necessary stubs will have been created from the classes designed using TDD to compile and execute the tests.

Apply modularity criteria so that the test classes do not grow too large. Consider defining one or more *fixtures*. Create one or more *suite*(s) to group test classes. Provide at least one *suite* that includes all the tests developed.

It is expected that the result of running the tests in this practice will be **red**. Therefore, if any implementation of the stubs has been done, it should be a fake implementation (*fake implementation*) so that **tests do not produce**

[10] Red (fail) and green (success) refers to the code of colours used by automated testing tools.

errors but fail. The objective is that all the implemented tests fail, except for exceptional cases clearly indicated and commented. In such cases, `fail` will be written as the last test instruction to achieve failure.

Tests will be categorised using the `@Category` annotation.

Purely P3:

The next phase of the TDD cycle should be conducted to obtain test successes (Green phase).

The coverage level achieved will be measured. In addition to the tests developed in the first phase, white-box tests will be developed to increase the coverage. The coverage level should be improved with as few tests as possible. The static rate *code to test* and the cyclomatic complexity of the implemented classes should be measured.

One of the classes implemented will be tested in isolation (the Newsletter shall be isolated from the News) using *mock objects* (optionally based on EasyMock or Mockito).

The implementation will be accomplished by applying *pair programming*, continuous integration and automatic dependency management.

Version control will be provided using `git`. `GitLab` on-premises (gitlab.inf.uva.es) will be used as a centralised repository and `Bitbucket` as a mirror. At the submission deadline, the instructor will be added to the project (hence no more commits and pushes to the centralised repository in `GitLab` or the backup in `Bitbucket` are allowed) as indicated in the submission rules.

The repository will not hold any `.class` or `.jar` or class documentation that can be generated at any time with *javadoc* (use `.gitignore`).

The use of `git` for branching and merging will be assessed. The branches of the project will include: the *master* branch, a *develop* branch, and one **branch-per-task**.

Pair programming techniques will be applied, so each task will be assigned to the pair, but the role of "*driver*" and "*observer* or *navigator*" should be interchanged along with the task development. The history of commits made in each branch will be reviewed and it will be assessed whether commits are made by the different members of the pair, as this will indicate their role as the driver at that moment. To do so, when working as driver, each developer will use his/her `git` user in his/her local workstation to commit to the task branch.

When a task is completed, it should be integrated into the *develop* branch. The integrations in *master* should only be of tested functional parts, with the tests succeeding. To achieve this, tests in isolation will be used when necessary.

The project will be automated using `ant` and `maven`.

As mentioned before, the repository will not hold any `.class` or `.jar`, but it should be enough to make a *pull* or *clone* from the `GitLab` repository in a fresh environment and use the `ant` script (`build.xml`) to compile, execute, run the tests, analyse the coverage, etc. Therefore, the external dependencies of the project will be managed by `maven`. The `ant` script will basically be used to delegate to `maven` in order to avoid having to memorise the `maven` tasks and parameters needed for each target.

The project will be based on the `maven` archetype maven-archetype-quickstart. The basic version of the `pom.xml` obtained from the indicated archetype should be reviewed and modified to add dependency management and update versions, as well as the configuration of plugins needed for coverage analysis and reporting, quality analysis and reporting, and any others needed by the authors.

The `ant` script (`build.xml`) must have at least the following targets that must be named exactly as follows:

compile: ensures that all dependencies are obtained and generates the `.class` from the source code (depends on the clean target). This is the default target.
runTDDAndBlackBoxTests: run the tests categorised as TDD and black-box excluding sequence tests.
runSequenceTests: run only sequence tests.
runWhiteBoxTests: run only the tests categorised as white-box, added to improve coverage level.
runAllTestsNoIsolation: equivalent to running the tests described in the previous three targets.
runAllTestsInIsolation: run all the tests in isolation (based on mock objects).
coverageReport: analyse test coverage obtained with the tests that target runAllTestNoIsolation runs and get different types of coverage reports (instruction coverage, branch coverage, complexity coverage).
genDocumentation: generate project documentation, including the doc folder generated with javadoc from self-contained documentation in the source code.
clean: clean the project space, remove any generated file and folder as a result of any other previous ant target execution.
measures: get a report with measures regarding code to test rate and the cyclomatic complexity of the implemented classes.

As mentioned before, to achieve these targets, it will be advisable to rely on calls to `maven`.

In `GitLab`, the continuous integration mechanism based on yaml syntax will be used. The pipeline phases will include works to build, run tests (with and without isolation), and a project deployment simulation that includes the documentation generation. The yaml file provided in the example published in `GitLab` will be used as a basis.

The result of the tests' execution in this phase should be **Green** in all cases, both in the tests in isolation based on mock objects and in the tests with real objects.

Tests shall be categorised using the annotation `@Category` to indicate their nature, tests used for TDD, black-box tests, sequence tests, tests in isolation based on mock objects, as well as white-box tests introduced to increase the level of coverage. It should be noted that several categories can be assigned to a test class (applying to all tests in the class) or to an individual test.

A.2.2 Instructions given for P3-sonarqube treatment

In this task, the same instructions are given as in the P3-core.

Modifications from the instructions given in the P3-core:

Replace "the Newsletter shall be isolated from the News" by "the Bus Network shall be isolated from the Line".

Add-on to the instructions given in P3-core for introducing `SonarQube` and its integration in CI/CD:

"In `GitLab`, the continuous integration mechanism based on yaml syntax will be used. The pipeline phases will include works to build, run tests (with and without isolation), perform quality analysis using SonarQube, and a project deployment simulation that includes the documentation generation. The yaml file provided in the example published in `GitLab` will be used as a basis. In the reference yaml file, the pipeline explains how to launch an analysis with SonarQube that is registered in the server. Do this from time to time to keep track of the evolution of the quality of your project."

Change in the `ant` target:

measures: perform a quality analysis of the project using SonarQube and obtain a report that measures the code to test rate and the cyclomatic complexity of the implemented classes.

A.2.3 Instructions given for P3-gamification treatment

Modified from the instructions given in the P3-sonarqube:

Replace "the Bus Network shall be isolated from the Line" by 'the Contest shall be isolated from the Ranking".

No further addition or modification regarding P3-sonarqube. Gamification is not included in this programming task. Nevertheless, the students are already aware that the contest will take place in P4-gamification. They know the prize and the rules that will be applied.

A.2.4 Instructions given for P4-core treatment

In this task, the same instructions are given as in the P3-core, but the references to pair programming are suppressed because it is an individual task.

Modifications from the instructions given in the P3-core:

Remove from the P3-core instructions the paragraph starting from "*Pair programming*".

Replace "Newsletter shall be isolated from News" by "Subject shall be isolated from Test".

Add-on to the instructions given in the P3-core:

"Complete the full TDD cycle Red-Green-Refactor. Emphasise in the last step: remove duplications, detect refactoring opportunities, improve the underlying design and code by refactoring.

Perform as long as possible the refactoring operations with the Eclipse Refactoring Plugin. The Refactor menu in Eclipse contains the option History which allows the performed refactoring operations to be seen.

Check that the Refactoring history is held in the `git` repository to keep track of the refactoring operations performed. Hence, in a fresh environment, a pull or clone from the version control repository will allow to see the refactoring history to be seen. To do this, add the hidden folder `.refactoring` to the version control. Eclipse stores the refactoring history in that folder. For further details, please refer to the course documentation.

At the end of this phase, it must be ensured that the result of running the tests is still Green."

A.2.5 Instructions given for P4-sonarqube treatment

In this task, the same instructions are given as in the P4-core.

Modifications from the instructions given in the P4-core:

Replace "Subject shall be isolated from Test" by 'BuddyQueue shall be isolated from Person".

Rewrite the first paragraph of **the add-on** in the P4-core to introduce `SonarQube`:

"Complete the full TDD cycle Red-Green-Refactor. Emphasise in the last step: remove duplications, detect refactoring opportunities, improve the underlying design and code by refactoring, follow the recommendations for quality improvement that you will find when running analyses with SonarQube."

A.2.6 Instructions given for P4-gamification treatment

In this task, the same instructions are given as in the P4-sonarqube.

Modifications from the instructions given in the P4-sonarqube:

Replace "BuddyQueue shall be isolated from Person" by "Subject shall be isolated from Test".

Add the contest rules and prizes as explained in Sec. 3.5.3.

Source Files (word or latex)

This piece of the submission is being sent via mail.

CRediT author statement

Yania Crespo González-Carvajal: Conceptualisation, Methodology, Data Curation, Formal Analysis, Writing- Original draft preparation, Writing - Review & Editing. **Carlos López Nozal**: Methodology, Formal Analysis, Data Curation, Visualisation, Writing - Review & Editing. **Raúl Marticorena Sánchez**: Validation, Writing - Review & Editing, Visualisation. **Margarita Gonzalo Tasis**: Validation, Writing- Original draft preparation, Writing - Review & Editing. **Mario Piattini Velthuis**: Methodology, Writing - Review & Editing, Supervision.

Declaration of Interest Statement

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: