POLITECNICO DI TORINO
Repository ISTITUZIONALE

Gamifying Testing in IntelliJ: A Replicability Study

Original
Gamifying Testing in IntelliJ: A Replicability Study / Straubinger, P., Fulcini, T., Garaccione, G., Ardito, L., Fraser, G.. - In: PROCEEDINGS OF THE ACM ON SOFTWARE ENGINEERING. - ISSN 2994-970X. - 2:ISSTA(2025), pp. 2407-2429. [10.1145/3728983]

Availability:
This version is available at: 11583/3000133 since: 2025-05-14T14:24:35Z

Publisher:
ACM

Published
DOI:10.1145/3728983

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript, con Copyr. autore

© Straubinger, Philipp; Fulcini, Tommaso; Garaccione, Giacomo; Ardito, Luca; Fraser, Gordon 2025. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in PROCEEDINGS OF THE ACM ON SOFTWARE ENGINEERING, http://dx.doi.org/10.1145/3728983.

(Article begins on next page)

26 November 2026

arXiv:2504.19294v1 [cs.SE] 27 Apr 2025

Gamifying Testing in IntelliJ: A Replicability Study

PHILIPP STRAUBINGER*, University of Passau, Germany
TOMMASO FULCINI*, Politecnico di Torino, Italy
GIACOMO GARACCIONE, Politecnico di Torino, Italy
LUCA ARDITO, Politecnico di Torino, Italy
GORDON FRASER, University of Passau, Germany

Gamification is an emerging technique to enhance motivation and performance in traditionally unengaging tasks like software testing. Previous studies have indicated that gamified systems have the potential to improve software testing processes by providing testers with achievements and feedback. However, further evidence of these benefits across different environments, programming languages, and participant groups is required. This paper aims to replicate and validate the effects of IntelliGame, a gamification plugin for IntelliJ IDEA to engage developers in writing and executing tests. The objective is to generalize the benefits observed in earlier studies to new contexts, i.e., the TypeScript programming language and a larger participant pool. The replicability study consists of a controlled experiment with 174 participants, divided into two groups: one using IntelliGame and one with no gamification plugin. The study employed a two-group experimental design to compare testing behavior, coverage, mutation scores, and participant feedback between the groups. Data was collected through test metrics and participant surveys, and statistical analysis was performed to determine the statistical significance. Participants using IntelliGame showed higher engagement and productivity in testing practices than the control group, evidenced by the creation of more tests, increased frequency of executions, and enhanced utilization of testing tools. This ultimately led to better code implementations, highlighting the effectiveness of gamification in improving functional outcomes and motivating users in their testing endeavors. The replication study confirms that gamification, through IntelliGame, positively impacts software testing behavior and developer engagement in coding tasks. These findings suggest that integrating game elements into the testing environment can be an effective strategy to improve software testing practices.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Integrated and visual development environments**.

Additional Key Words and Phrases: Gamification, IDE, IntelliJ, Software Testing, Replicability Study

ACM Reference Format:
Philipp Straubinger, Tommaso Fulcini, Giacomo Garaccione, Luca Ardito, and Gordon Fraser. 2025. Gamifying Testing in IntelliJ: A Replicability Study. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA106 (July 2025), 23 pages. https://doi.org/10.1145/3728983

*Both authors contributed equally to this research.

Authors' Contact Information: Philipp Straubinger, University of Passau, Passau, Germany, philipp.straubinger@uni-passau.de; Tommaso Fulcini, Politecnico di Torino, Torino, Italy, tommaso.fulcini@polito.it; Giacomo Garaccione, Politecnico di Torino, Torino, Italy, giacomo.garaccione@polito.it; Luca Ardito, Politecnico di Torino, Torino, Italy, luca.ardito@polito.it; Gordon Fraser, University of Passau, Passau, Germany, gordon.fraser@uni-passau.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2994-970X/2025/7-ARTISSTA106
https://doi.org/10.1145/3728983

1 Introduction

In software development, ensuring the quality and reliability of software products is a key priority. Software testing plays a crucial role in this process, serving as a core element of the software development lifecycle [37]. It helps identify defects, improve functionality, and ensure that applications meet their intended requirements. Despite its importance, however, software testing is often seen as one of the less appealing phases of development, which can lead to testers feeling undervalued or disengaged [33]. This lack of motivation can hinder productivity and negatively impact the overall quality of the software produced [24].

To address these challenges, recent literature has explored the concept of *gamification*, a technique that applies game-design elements in non-game contexts [12]. Across various aspects of Software Engineering gamification has shown promising results in terms of engagement and performance [15]. There have been several attempts to incorporate gamification into the software testing process, both to enhance learning (especially in academic settings) and to improve the practical execution of testing. Gamified tools and environments have been applied to a range of testing activities, including unit testing and GUI testing [7, 14, 23], test creation, execution, and maintenance [29, 35, 36].

Most studies in this area have focused on exploring new tools and conducting preliminary evaluations to validate their effectiveness and user experience (UX) in small-scale contexts [15]. So far, these gamified approaches have largely been developed independently, with limited efforts to generalize the results across different contexts. However, there is a need for consolidation of the findings, requiring an effort to replicate the existing validation studies in other contexts, to achieve higher confidence in the obtained results and move towards well-established understanding.

With this goal in mind, we conducted a replication study to assess one of the latest gamified tools for unit testing: IntelliGame, a gamified plugin for the IntelliJ IDEA Integrated Development Environment (IDE) that rewards testers with achievements for good testing practices [36]. Given the promising results of the initial evaluation, this tool merits further exploration, as limited participation may restrict the generalizability of observed benefits.

This paper presents our attempt to adapt IntelliGame for broader use, validating it through a controlled empirical experiment aimed at generalizing its effectiveness. The key contributions of this paper are as follows:

- We present an empirical experiment to validate the existing IntelliGame tool using a two-group experimental design on a new programming language with a sample of 174 participants.
- We describe a statistical analysis of participants' behavior and performance in writing and executing tests.
- We present a comparison of the findings from the original study with those of our study.

Our replication confirms that gamification can effectively impact user behavior by encouraging the creation of more tests, more frequent test executions, and increased use of testing tools, like coverage reports and debugging tools. We observed that while both groups exhibited similar testing approaches, gamified participants wrote more tests, focused on finer details, and produced code with fewer failing tests of the reference test suite—implying a slightly better functional outcome. Additionally, we found that achievement levels in IntelliGame correlated positively with both tester motivation and the quality of test suites.

These results align with several conclusions from the original study, specifically that gamification can increase testing frequency and improve user engagement with tools for quality assurance. However, our study adds nuances. For instance, we found that IntelliGame also impacts debugging behavior—a result not highlighted in the original study. Further introspection on test quality and individual achievements indicates that while gamification users created more detailed test

cases, their increased focus on quantity occasionally led to higher failure rates and test smells, particularly in the area of date and time handling. This indicates that while gamification encourages thoroughness, the IntelliGame approach would benefit from additional or modified achievements aimed at quality to balance the focus on test quantity. Further, unlike the original study, our participants in the *treatment* group reported greater time pressure and showed a greater awareness of the quality of their tests compared to the *control* group, likely due to the continuous feedback provided by IntelliGame.

In summary, we successfully replicated the original study on a gamification plugin with achievement features in the integrated development environment, largely confirming its results.

2 Background

2.1 Gamification in Software Testing

Gamification is a technique adopted for increasing motivation and interest in unappealing tasks and is considered as the usage of game elements in non-recreational contexts [12]. An additional benefit is that tasks performed in a gamified environment can produce better output compared to non-gamified equivalent tasks [11, 32]. Among the various game elements, the most commonly adopted ones in gamified approaches are points, leaderboards, badges, and awards [1, 9], with other possible elements such as avatars (a user's visual representation in the gamified system), progress bars (indicators of a user's progression toward completing a task), feedback (the game's reaction to the user's tasks and actions), achievements (specific rewards earned after completing tasks multiple times), and penalties (detrimental effects being applied after incorrect behavior, to deter further errors) being used depending on the application context.

Using gamification in software testing education has proven effective, boosting student motivation and engagement in what is often seen as a less appealing subject [10, 40]. Gamified methods have been successfully integrated into various aspects of software testing, with unit testing being a common area of focus [9, 15]. Other areas that have seen gamification benefits include introductory courses [2, 31], testing tools [6, 25], exploratory [8, 20, 34], mutation [29], and GUI testing [5, 17].

For instance, gamification was introduced into a software testing course [3], comparing student performance with those from a prior cohort using a traditional teaching approach. The gamified course led students to run more test suites and uncover more bugs than the non-gamified course. Similarly, GUI testing [7] was enhanced through coverage indicators, leaderboards, and scoring, comparing students in gamified and non-gamified groups. The gamified group showed greater page coverage and included more assertions in their test cases, highlighting gamification's impact on engagement and thoroughness in testing.

2.2 IntelliGame

IntelliGame is a plugin for the popular IntelliJ IDEA,[1] an Integrated Development Environment (IDE) that supports development in Java, Kotlin, and JavaScript. This plugin integrates gamification into the testing process through a straightforward yet effective game element: achievements [36]. IntelliGame tracks various testing-related activities within the IDE, such as running tests, using assertions, debugging, and improving test coverage. The system monitors user actions and awards good behavior with achievements based on predefined criteria. The implementation leverages IntelliJ's event system using a publisher-subscriber pattern, ensuring accurate tracking of testing behaviors. The design ensures minimal workflow disruption by allowing developers to engage with gamification at their own pace. Progress is displayed through a user interface with trophies and progress bars (Fig. 1), while notifications provide real-time feedback on achievements.

[1] https://www.jetbrains.com/idea/

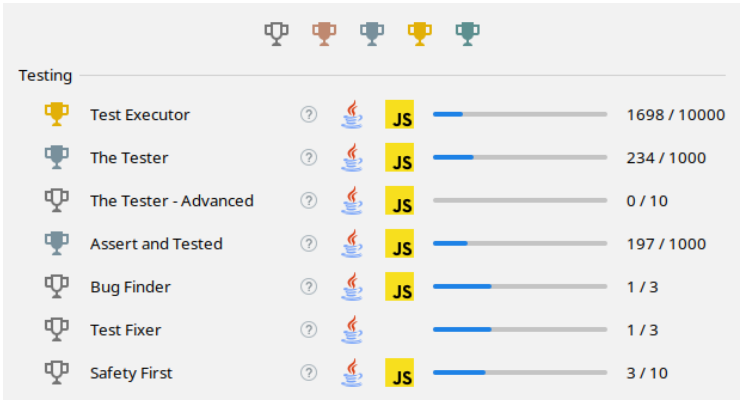


Fig. 1. IntelliGame window showing part of the achievements and their progress

IntelliGame offers a set of achievements across four key areas: testing, coverage, debugging, and test refactoring achievements:

- **Testing achievements** reward developers for writing and running tests, encouraging frequent test execution, assertion triggers, and addressing failing tests.
- **Coverage achievements** motivate testers to use test coverage tools and improve coverage with each new test run.
- **Debugging achievements** recognize developers using the debugger of IntelliJ to fix bugs and resolve code issues.
- **Test Refactoring achievements** incentivize developers to enhance existing test code, such as by refactoring redundant code into helper functions.

Achievements are represented by different kinds of trophies indicating the tester's current level, a name, a brief description visible on hover, and a progress bar showing progress toward the next level. Achievements provided by IntelliGame mimic important testing milestones, such as reaching a certain line, methods, and branch coverage running tests, or reward positive testing behavior such as measuring coverage or fixing tests. As soon as testers meet the requirements for a certain achievement, they are notified via a notification system inside the IDE. Figure 1 illustrates the plugin's achievements window with all available achievements displayed.

Each achievement consists of multiple levels, with progress earned through actions such as running tests, using assertions, debugging, and improving test coverage. These achievement levels create a structured progression system that rewards developers for engaging in various testing activities. The boundaries for each level were set based on an analysis of typical developer workflows in a pilot study before the original study [36], ensuring a balance between accessibility for beginners and meaningful challenges for experienced users. Levels unlock once these predefined thresholds are met, encouraging continuous learning and skill refinement in a motivating and structured way.

The intended use case scenario for IntelliGame is mainly to support developers during their implementation and testing tasks, for example when onboarding new developers, or for improving quality-related habits for more experienced developers. Developers, while developing their codebase and writing and running unit tests should engage with the achievements, rewarding when these tests pass, when they are run with coverage, and when they are maintained by fixing code for failing tests. This scenario is particularly suitable for Test-driven development, where the coding is based on iterative refinement from a given test suite. It is also conceivable to integrate the plugin into academic courses to provide students with feedback while learning testing concepts.

IntelliGame has been previously validated in a controlled experiment with 49 participants [36]. This first validation had the objective of assessing whether the tool can influence testing behavior and the quality of the resulting test suites and codebases. The participant's performance was measured in terms of test quality (number of tests written, code coverage, mutation scores), code functionality (number of tests passing against a reference test suite), and user experience. To study the effects of these metrics, the participants were divided into four groups, each with a different treatment, and asked for the same implementation task: developing two Java functions and verifying the correctness of their implementation. The different treatments consisted of (1) a version of the gamification plugin providing only notifications corresponding to milestones, but no achievements to the users, (2) the IntelliGame plugin itself as previously described considering it as the treatment group, (3) a group using the gamification plugin but explicitly indicating to try to maximize the levels of the achievements and (4) a control group with a plugin with no effects, just collecting data. Further information and more detailed explanations regarding IntelliGame and this study are contained in the original study [36].

2.3 Replicability, Reproducibility, and Repeatability

Empirical science is a cornerstone of the scientific method, as practical validation of theories is essential for scientific progress [19]. Trust and reliability in empirical findings are critical to advancing theoretical and technical fields, with independent reproducibility being a fundamental requirement for accepting any empirical result as established [30].

Based on the ACM Replicability guidelines [13], there exists different concepts and granularities:

- **Repeatability** is the ability of the original research team, under identical conditions, to confirm the experiment's results across multiple trials.
- **Reproducibility** occurs when a different team, following the same measurement procedures, can achieve the same results and precision under identical operating conditions, regardless of location.
- **Replicability** is the ability of an independent team to obtain the same results and precision in a different location, using a different measurement system, across multiple trials.

The scientific community should ideally aim for the highest possible levels. Our goal is to conduct a replicability study: As an independent team, we replicate the experiment with a similar, yet different experimental setup. The main differences are (1) a different and larger population of participants, (2) a different experimental object, with several non-trivial functions to implement, (3) a different target programming language, and (4) new research questions aiming to shed further light on the specific details and influences of the gamification aspects of IntelliGame.

3 A Replicability Study of the Initial IntelliGame Study

In this section, we outline the experimental design according to guidelines for reporting software engineering experiments [18]. To ensure the validity of this description, we also used the checklist from the Empirical Standards for Software Engineering Research [27].

3.1 Research Questions

The experiment aims to assess the impact of IntelliGame on its users. To achieve this, we established goals addressing different potential effects of IntelliGame on users, each linked to a specific research question (RQ). Since the primary objective is to replicate an existing study, we adopted the original research questions from the initial validation paper [36]:

- **RQ1**: Does IntelliGame influence testing behaviour?
- **RQ2**: How does IntelliGame influence resulting test suites?

- **RQ3**: Do achievement levels reflect differences in test suites and activities?
- **RQ4**: Does IntelliGame influence the functionality of the resulting code?
- **RQ5**: Does IntelliGame influence the developer experience?

The goals are as follows: first, to understand IntelliGame's influence on testing behavior and test quality; second, to evaluate whether achievement levels within IntelliGame affect the test suite; third, to assess the functionality of the assigned code base; fourth, to explore users' overall experience with IntelliGame. One aspect not considered in the original study is the quality of the tests and the importance of individual achievements themselves. Therefore, we add two new research questions to investigate test quality and achievements:

- **RQ6**: Do users of IntelliGame write high-quality tests?
- **RQ7**: What are the most important achievements?

3.2 Object Selection

The experimental object needed for the experiment had to be a real-world project to be used as a reference to assign students some programming tasks. To select a suitable project for our needs, we began exploring a well-known database of JavaScript libraries in March 2023.[2] We established specific inclusion criteria (IC) to determine if a project was eligible to be used as the experimental object. The inclusion criteria were as follows:

- The project must be an open-source artifact.
- The project must be implemented in JavaScript or TypeScript.
- The project should be a standalone library containing functions related to data types, easily understandable by students, rather than a development framework (e.g., Vue, or Angular).
- The project must include unit tests to verify the correctness of all its functions.
- The project documentation should be clear and comprehensive, detailing each function.
- The project must be well-maintained, with its last update occurring within three months.

We started browsing the database, focusing on the *miscellaneous* category to find a project that met our criteria. We applied the inclusion criteria iteratively to each project on the list, which was ranked by popularity. After filtering, we identified Date Fns[3] as a suitable choice.

The original Date Fns project features nearly 250 utility functions related to date data types. These functions range from simple tasks, such as comparing two dates, to more complex operations like transforming a date to conform to ISO or RFC standards.[4] This library is highly popular, used in millions of projects, and is well-maintained by more than 390 contributors, with over 1000 forks.

Since the original project utilized a different testing framework than the one the participants were familiar with, we needed to modify the configuration to suit our needs. Specifically, we manually converted the original test suite Ts from the Vitest[5] framework to the equivalent Jest format Ts'. To ensure that our refactoring process did not introduce any errors, we executed the original and the refactored test suites, confirming that both produced no failures and the same mutation scores.

3.3 Pilot Study

Before conducting the main experiment, we carried out a pilot study to improve and refine our methodology. In 2023, we organized a preliminary attempt at Politecnico di Torino to replicate the original experiment, confirm the feasibility, and fine-tune our approach. In the pilot, we adapted

[2] https://web.archive.org/web/20230322130719/https://www.javascripting.com/
[3] https://github.com/date-fns/date-fns
[4] ISO 8601 formats dates as year-month-day hour:minute:second.millisecond (e.g., 2022-09-27 18:00:00.000). The RFC standard includes the weekday, day, month, year, time in 24-hour format, and timezone code, such as Tue, 27 Sep 2022 18:00:00 EST.
[5] https://vitest.dev/

IntelliGame to support JavaScript and TypeScript, using Jest as the testing framework. Out of the original 26 Java-based achievements, 19 were successfully implemented for JavaScript, covering testing, coverage, and debugging-related achievements.

Moving the focus of IntelliGame from Java to applications to JavaScript or TypeScript could have been done in two ways: by migrating the underlying IDE from IDEA to WebStorm, the web-specific IDE from JetBrains, or by adding support for a different testing engine, namely Jest, in addition to JUnit. Since readjusting the plugin both to a different IDE and a different testing framework would require significantly more effort than simply adding the support for Jest, we decided to pursue the second path. In case a positive effect of gamification is measured also for testing in TypeScript, further research endeavors will be directed towards the complete porting of the plugin into WebStorm.

The large-scale pilot study with 152 participants validated our methodology and highlighted areas for improvement, particularly regarding task complexity relative to time constraints. During the pilot, participants were initially tasked with implementing 23 JavaScript functions, but most could not complete half within the allotted time. Consequently, we scaled down the main experiment to 11 functions to better fit within time limitations. Additionally, the object of study—Date-Fns, a JavaScript library—remained consistent across the pilot and main experiment. The 2023 course required students to complete a JavaScript group project, necessitating adjustments to convert the TypeScript-based Date-Fns project to JavaScript. However, in 2024, the course project shifted to TypeScript, easing compatibility and familiarity with Date-Fns for students.

Another critical insight from the pilot was the need for separate experimental sessions. In mixed sessions, students could identify their group assignment, which may have influenced performance. To counteract this, the main experiment hosted distinct sessions for each group, preventing participants from inferring differences in treatment.

For a complete description of the pilot study methodology and findings, see [16].

3.4 Experiment Design

The primary objective of this study is to investigate how gamification influences users' testing behavior. To achieve this, we conduct a simple two-group experiment with different participants. The experiment involves using two versions of IntelliGame: one with gamification (the *treatment* group) and one without (the *control* group). The *treatment* group receives the adapted version of IntelliGame, while the *control* group is provided with a plugin that only collects data. Both plugin versions calculate the levels and store the same data. However, only the plugin in the treatment group displays achievements, shows progress, and sends notifications. Although the original experiment used four groups for the comparisons, the main analysis consists of comparing the treatment with the control group, which we focus on in our replication.

The controlled experiment involved a programming task in which participants were asked to develop code based on specific requirements and ensure its correctness. They are not given any restrictions on how to test their code. We base the programming assignment on code extracted from an existing software project, similar to the approach taken in the original experiment, to enhance the ecological validity of the study. The task is designed to be challenging yet feasible within the allotted 150 minutes, both in terms of implementation and testing.

Participants are given a simplified version of the GitHub Date Fns project. We carefully selected 11 functions from the original project, prioritizing simplicity in terms of complexity and the number of code lines. This was intended to ensure that most participants could successfully develop and test these functions without feeling overwhelmed.The functions to be developed and tested include boolean functions checking date values such as *isAfter*, *isPast*, *isWeekend*, getters for some values from the date value such as one that returns the number of days in a month of the given date, or the

day of the month of a given date, namely *getDaysInMonth* and *getDate*. Another function required was to perform a sum of days to a given date: *addDays*.

The bodies of the functions are left empty, retaining only the requirement descriptions, function names, and parameters. The project includes complete documentation for the date class and functions, preventing participants from accessing online references or external suggestions. To verify the correctness of their implemented code, we provide two methods: a `main.ts` file that mirrors the setup of the original experiment and functions similarly to the Main class in Java, along with a Jest configuration containing blank test files for participants to complete and execute.

After the implementation phase, participants completed an online exit survey. The survey began with demographic questions about their studies, age, gender, and experience with TypeScript and Jest. The second page included general questions about the implementation and testing of the class. A third page, shown only to the *treatment* group, gathered feedback on their experience with the plugin. Responses were based on a five-point Likert scale, using questions from the original study for consistency [36]. An optional free-text field allowed participants to add additional comments.

3.5 Participant Selection

We selected a sample of master's degree students because they closely represent new hires undergoing onboarding [21] and because this group allowed us to recruit a large sample size effectively.

To recruit participants, we issued a call for volunteers during a Software Engineering course at Politecnico di Torino, where the experiment was conducted. We used an eligibility survey to gather applicants, which included basic demographic questions and information about their familiarity with the relevant work environment. Students were encouraged to participate in the study, with an additional course grade point offered upon successful completion. Success was based on participation and commitment rather than correctness or performance, with students free to withdraw from the experiment at any time.

We received 264 survey responses, with 232 students actually participating in the experiment. However, only 218 provided usable data, and of these, only 174 participants' projects included the `TestReport` from IntelliGame. IntelliGame generates this report whenever the program is executed, debug mode is used, or tests are written or executed. Incomplete data was largely due to setup issues, failure to install the plugin, or lack of actual participation.

The participants were predominantly in their early twenties, with an average age of 23.4, and about one-quarter were female. All were students, though some had industry experience, albeit in the minority. Most had less than six months of experience with JavaScript and TypeScript, and nearly all had less than three months of experience with Jest. All students were actively enrolled in the Software Engineering course and were working on a Node.js project that involved testing, where they became familiar with the experimental framework, including the programming language, testing, and build environment. Testing experience for the students was limited to the Software Engineering course, however, at the time of the experiment the students had been actively engaged for a month in creating a test suite for their group project.

Unfortunately, most of the students participating in the experiment had no working experience in the field, therefore we cannot fully generalize the findings for experienced testers. To explore the impact of IntelliGame on developers and testers more comprehensively, one potential approach would be to separate the data from students who possess prior industry experience or those who are simultaneously employed and studying. However, this analysis goes beyond the scope of this experiment, as the small number of individuals fitting these criteria without our participant pool would necessitate data collection across multiple years of experimentation.

3.6 Experimental Analysis

The experiment's analysis compares metrics across two groups, the *control* group without the use of IntelliGame, and the *treatment* group using the adapted version of IntelliGame. To assess the significance of the differences between these groups, we use the exact Wilcoxon-Mann-Whitney test [22] to calculate p-values, applying a confidence threshold of $\alpha = 0.05$.

3.6.1 RQ1: Does IntelliGame influence testing behaviour? We analyze testing behavior based on the number of (1) tests written, (2) tests run, (3) tests run with coverage enabled, and (4) instances of using debug mode to execute tests.

3.6.2 RQ2: How does IntelliGame influence resulting test suites? This research question focuses on assessing the quality of the test suites by measuring (1) the number of tests, (2) test coverage on both line and branch levels, and (3) the mutation score of the final test suites. The number of tests and test coverage are automatically collected and calculated during test execution using the Jest framework.[6] Mutation testing is conducted with StrykerJS (Stryker Mutator),[7] a popular mutation testing engine in web development. To ensure accuracy, failing tests were excluded from the mutation analysis, as Stryker requires a fully passing test suite.

3.6.3 RQ3: Do achievement levels reflect differences in test suites and activities? During the experiment, IntelliGame logged each user interaction, recording both the current levels achieved and the status of each achievement. We calculate the Pearson correlation [26] between these achievement levels and metrics such as line coverage, branch coverage, mutation score, and the number of tests to compare the two groups.

3.6.4 RQ4: Does IntelliGame influence the functionality of the resulting code? To evaluate code functionality, we use the original project test suite—comprising 60 test cases that cover 95.83% of the original implementation's lines—as the ground truth. We execute these tests on each participant's final code version, comparing the number of passing and failing tests between both groups. Additionally, we assessed all intermediate code versions using the commit history.

3.6.5 RQ5: Does IntelliGame influence the developer experience? To answer this research question, participants completed an exit survey that included a series of 5-point Likert scale questions designed to assess their perceptions of the tasks and the approaches they used. Additional questions were presented to the *treatment* group specifically to gauge their impressions of IntelliGame.

3.6.6 RQ6: Do users of IntelliGame write high-quality tests? To address this research question, we examine the test suites created by participants using the test smell [38] detection system Smelly Test[8] and report the ratio of test smells to tests. Smelly Test incorporates a total of eight different test smells for TypeScript, including conditionals, timeouts, console printing, and empty tests. In addition, we execute the tests to identify any failing ones. A failed test indicates either that the participant did not complete the test initially or that the test lacks the robustness to remain reliable over time.

3.6.7 RQ7: What are the most important achievements? To answer this research question, we analyze the final `TestReport` from each project to assess participant progress in both groups. We then calculate the mean achievement progress for each group and compute the p-values to identify which achievements the participants aimed for.

[6] https://jestjs.io/
[7] https://stryker-mutator.io/
[8] https://github.com/marabesi/smelly-test/

3.7 Threats to Validity

This subsection discusses potential validity threats in our study, following the classifications by Wohlin et al. [39], and our approaches to mitigate these risks.

3.7.1 Threats to Conclusion Validity. Threats to conclusion validity involve factors that may impair our ability to correctly infer relationships between the treatment and the study outcomes. To address these threats, we used the Wilcoxon-Mann-Whitney test with a significance level of $\alpha = 0.05$ for statistical analysis. Our data was gathered and calculated automatically with tools like Jest for test execution, which minimizes human error during data collection. Participants were randomly assigned to one of two groups, each provided with a different operational environment. All participants were university students enrolled in two Software Engineering courses taught in different languages but covering the same English-language material. Thus, we did not consider course language a confounding factor. While we have no evidence of varying skill levels between groups (all participants had the foundational knowledge to complete the experiment, as part of their course), we acknowledge that a different group assignment could have produced varied statistical outcomes. However, given the large sample size and knowledge distribution, this threat is unlikely, though not entirely dismissible.

3.7.2 Threats to Internal Validity. Threats to internal validity involve factors that may impact the causal relationship between the independent and dependent variables. The experiment took place during regular class time, with participants using their laptops in a familiar classroom setting. While the experiment duration of 150 minutes could have led to fatigue, participants were allowed short breaks, and no one dropped out during the sessions. Another potential threat is the unauthorized use of external tools (e.g., GitHub Copilot or ChatGPT). To mitigate this, we explicitly instructed participants not to use such tools, since task completion time and performance were not part of the evaluation, and monitored them throughout the session. However, sporadic use of these tools cannot be entirely ruled out. Since this was a between-subjects experiment, learning biases were absent by design. Selection bias, while possible, was minimized by random group assignment and by rewarding participation rather than performance. We do not believe that the extra course grade incentive led to a disproportionate number of intrinsically motivated students. Finally, as we used previously validated tools (i.e., IntelliGame, Smelly Test, Stryker, and Jest), we consider the risk of instrumentation bias negligible.

3.7.3 Threats to Construct Validity. Threats to construct validity concern the generalizability of the experimental results to the theoretical constructs. One potential issue is that the original test suite used as ground truth may not cover all edge cases. However, since the test suite achieves 95.83% line coverage and 100% mutation score for the selected functions, we consider any remaining edge cases negligible for this experiment, where participants had only 150 minutes to complete and test their tasks.

3.7.4 Threats to External Validity. Threats to external validity refer to limitations in the generalizability of the experiment's findings. Our selection of participants was based on convenience, as recruiting a large student population was more feasible. Nonetheless, we argue that graduate students reasonably represent newly recruited employees [21]. A further iteration of this experiment can mitigate this threat by encompassing only students with working experience. The experimental object was a real software library, commonly used in practice, which supports our goal of generalizing the findings of the original study [36]. Our study complements previous research, expanding its findings to typical software development contexts.

4 Results

4.1 RQ1: Does IntelliGame influence testing behaviour?

Test Creation. IntelliGame aims to encourage the writing of unit tests, and it appears effective in doing so. In the *treatment* group, only 10.59% of participants did not write any Jest tests, whereas in the *control* group, this figure was higher at 21.34%. Although this difference is close to statistical significance (exact Fisher test [4], $p = 0.064$), it is not fully conclusive. Additionally, participants in the *control* group relied more often on the main method for manual testing: 73.03% of the *control* group used the 'main' method at some point for testing, compared to only 55.29% in the *treatment* group, which is a significant difference ($p = 0.018$).

Figure 2a illustrates the progression of Jest tests written over time. Both the *treatment* and *control* groups began writing tests early on, right from the start. However, the *treatment* group consistently wrote more tests from the outset and showed a sharper increase in test writing compared to the *control* group, starting around minute 39. From minute 53 until the end, the difference between the groups is statistically significant, as indicated by non-overlapping confidence intervals.

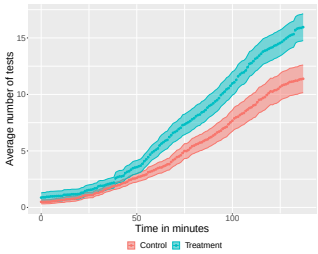
Test Executions. Figure 3a displays the number of test executions, with an average of 47.56 in the *treatment* group and 20.97 in the *control* group—a significant difference ($p < 0.001$). The *treatment* group also had a notably higher maximum number of executions, reaching 316 compared to just 80 in the *control* group. Additionally, Fig. 2b shows test executions over time, along with the 84.6% confidence intervals. A significant difference between both groups emerges at minute 56, indicating that the achievements in IntelliGame motivated participants to execute their tests more frequently.

Coverage Measurement. Figure 3b compares test executions with coverage collection enabled. In the *control* group, only 6.74% of participants used this IntelliJ feature, resulting in an overall average of 0.11 coverage executions per participant. In contrast, the *treatment* group used the coverage report more frequently, with an average of 2.24 executions per participant. Figure 2c shows coverage executions over time, revealing that the first participant in the *treatment* group used the coverage report from the very start, while the *control* group only began using it after 39 minutes. This indicates a significant difference ($p < 0.001$) from the beginning, as the *treatment* group participants demonstrated greater engagement with coverage collection.

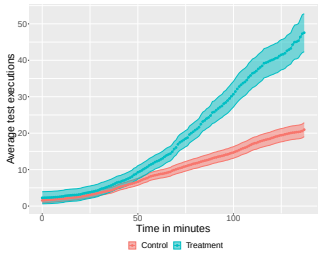
Debug Executions. Figure 3c shows the frequency of test or program executions in debug mode. Participants in the *treatment* group used debug mode an average of 1.96 times, compared to 1.06 times in the *control* group—a significant difference ($p = 0.015$). This difference is also evident in Fig. 2d, where confidence intervals diverge after minute 79. Some control group participants initially used debug mode instead of regular runs, explaining their early lead. Given the program's complexity, particularly in date and time handling, participants increasingly relied on debug mode in later stages, with achievements in IntelliGame likely encouraging more frequent use.

Summary (*RQ 1*): IntelliGame significantly influences participants' testing behavior: they prefer using Jest tests over main testing, create more tests, run their tests more frequently, make greater use of coverage reports, and rely more often on debug mode for both program and test debugging.

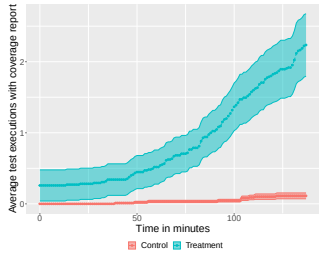
Our findings confirm the results of RQ1 from the original paper and additionally show that IntelliGame also affects debugging behavior.



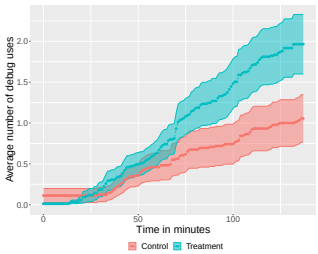
(a) Number of tests created over time



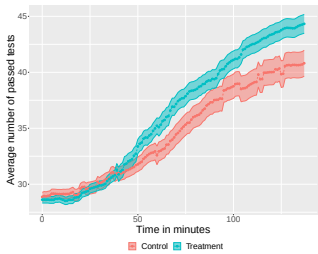
(b) Number of test executions over time



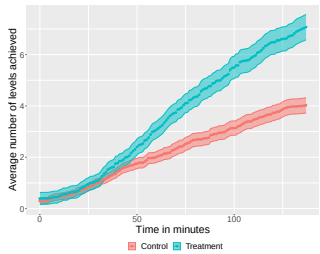
(c) Number of test executions with coverage over time



(d) Number of debug uses over time

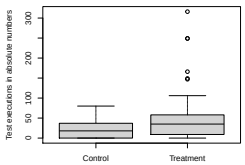


(e) Number of passed tests over time

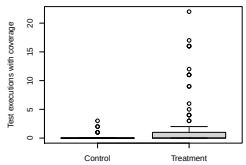


(f) Number of levels over time

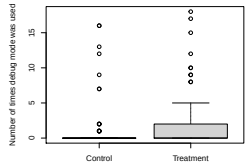
Fig. 2. Differences between *control* and *treatment* groups over time



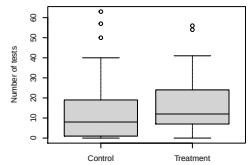
(a) Number of test executions during the experiment



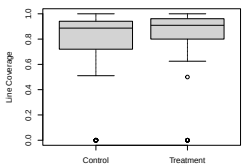
(b) Number of test executions with coverage in IntelliJ



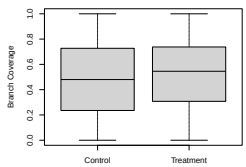
(c) Number of times the IntelliJ debug mode was used



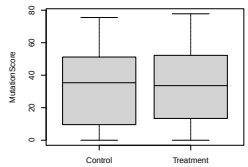
(d) Number of tests written by the participants



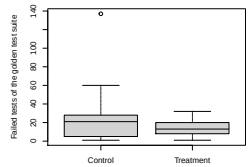
(e) Line coverage of the final test suites



(f) Branch coverage of the final test suites



(g) Mutation scores of the final test suites



(h) Number of failed tests according to the golden test suite

Fig. 3. Differences between *control* and *treatment* groups

4.2 RQ2: How does IntelliGame influence resulting test suites?

Number of Tests. Figure 3d compares the final test suites based on the number of Jest tests they contain. Participants in the *treatment* group wrote an average of 15.95 tests, whereas the *control* group averaged 11.38 tests. The exact Wilcoxon-Mann-Whitney test confirms that the *treatment* group wrote significantly more tests than the *control* group ($p = 0.003$). We found indication that they focused on more fine-grained testing, for example by using Boundary Value Analysis (BVA) [28] to test dates just inside or outside valid ranges. However, to some extent the increased test count may also simply be due to IntelliGame encouraging more tests, possibly including some redundant ones.

Code Coverage. Figure 3e compares line coverage between the *treatment* and *control* groups, showing an average coverage of 81% and 75%, respectively. Although the *treatment* group thus achieved higher coverage, this difference is not statistically significant ($p = 0.14$). In both groups, some participants achieved 0% line coverage, indicating they did not write any unit tests and relied on the main method for testing. Similar results are observed for branch coverage (Fig. 3f), with average coverages of 51% in the *treatment* group and 46% in the *control* group ($p = 0.34$). Overall, although the *treatment* group had slightly higher average coverage, the difference was not statistically significant. Given that participants received documentation of the methods to implement, achieving coverage was relatively straightforward, which may account for the lack of a significant difference in coverage compared to the original study.

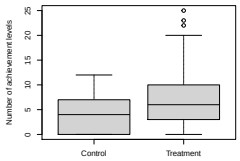
Mutation Score. The mean mutation score was 34.08% for the *treatment* group and 32.4% for the *control* group, a non-significant difference ($p = 0.58$) (Fig. 3g). This suggests that both groups approached test writing similarly, leading to comparable coverage and mutation scores. The mutation analysis excluded failing tests, which may explain the lack of significant difference in mutation scores. Upon investigating failing tests, we observed that many of them are related to date discrepancies, as we reran all tests for analysis in October 2024, about four months after the experiment. We will investigate these failing tests and test quality further in Section 4.6. However, to understand whether the lack of significant improvement of the mutation score is a result of the date difference in our analysis, we re-ran the mutation analysis with the system time being set to the time and date of the commit, and while the mutation scores increase for both groups, the difference remains non-significant.

Summary (*RQ 2*): IntelliGame led the *treatment* group to write significantly more Jest tests than the *control* group. However, since code coverage and mutation scores remained similar between the groups, this suggests that the tool may have encouraged quantity over effectiveness in test writing.

Our findings confirm that IntelliGame influences test suite outcomes by increasing the number of tests. However, we did not observe significant differences in code coverage or mutation scores, so we were unable to replicate the original study's results in these areas.

4.3 RQ3: Do achievement levels reflect differences in test suites and activities?

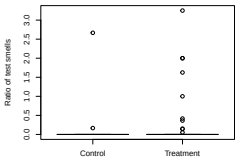
RQ1 demonstrated that participants in the *treatment* group are more actively engaged in testing activities compared to those in the *control* group. Figure 4a compares the achievement levels reached by participants in both groups, showing a mean of 4.02 for the *control* group and 7.07 for the *treatment* group. Figure 2f indicates that this difference becomes evident starting at minute 48



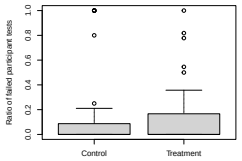
(a) The number of achievement levels reached by the participants

Variable	r^2	p-value
Line Coverage	0.42	< 0.001
Branch Coverage	0.38	< 0.001
Mutation Score	0.48	< 0.001
Number of Tests	0.65	< 0.001

(b) The Pearson correlations based on the number of levels



(c) The ratio of test smells in the tests of the participants



(d) The ratio of failed tests written by the participants

Fig. 4. The number of achievement levels, their Pearson correlations with different test suite metrics as well as the ratios of failed tests and test smells

of the experiment. The difference is significant, as confirmed by the exact Wilcoxon-Mann-Whitney test $p < 0.001$. Thus, developers who earn achievements are indeed more committed to testing.

To assess whether developers with higher achievement levels produce better test suites overall, Fig. 4b presents the Pearson rank correlations between test suite metrics (RQ2) and achievement levels. A strong significant correlation exists between the number of tests and achievement levels, along with a moderate positive correlation for both line coverage and mutation score. Additionally, there is a weak significant correlation between achievement levels and branch coverage. These findings strongly support the effectiveness of the gamification approach, demonstrating that earning achievements is associated with producing better test suites.

Summary (*RQ 3*): Higher achievement levels indicate greater motivation for testing and lead to better-quality test suites.

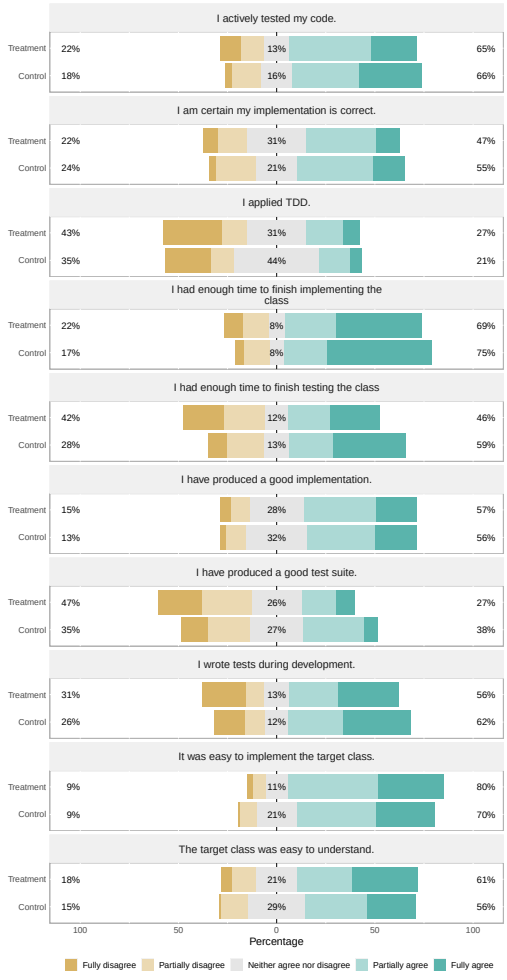
Our findings support those of the original paper, showing that higher achievement levels lead to increased motivation and improved test suites.

4.4 RQ4: Does IntelliGame influence the functionality of the resulting code?

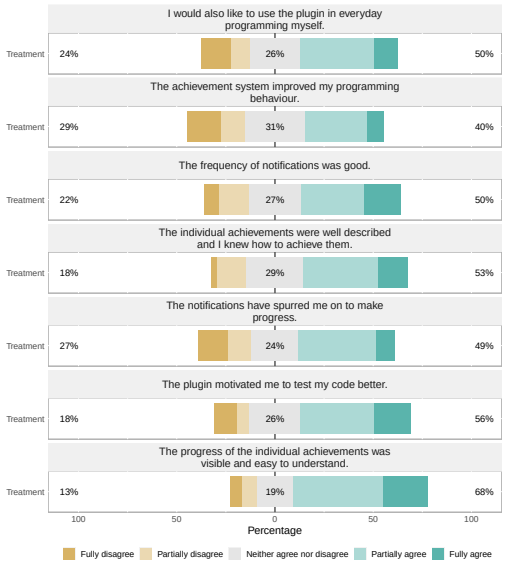
Figure 2e illustrates the number of passed tests in the golden test suite throughout the experiment, as a proxy for how much of the target functionality is already correctly implemented. Notably, the *treatment* group began passing tests earlier, starting at minute 50, compared to the *control* group. Although there is a small overlap around minute 90, the differences remain significant until the end of the experiment since the graphs do not overlap anymore. Since some implementations were provided, all participants had at least 28 passing tests out of the 60 in the golden test suite.

By the end of the experiment, the average number of failing tests in the golden test suite (Fig. 3h) was 14.32 for the *treatment* group and 19.25 for the *control* group, showing a nearly statistically significant difference ($p = 0.054$). This indicates that participants in the *treatment* group implemented more functionality correctly by the end of the experiment, while the *control* group had a higher number of errors in their implementations.

Summary (*RQ 4*): Users of IntelliGame demonstrated earlier passing and fewer failing tests in the golden test suite, indicating better implementation of functionality compared to the *control* group.



(a) General survey responses—ranging from negative on the left to positive on the right



(b) Answers on the IntelliGame plugin—ranging from negative on the left to positive on the right

Fig. 5. Survey responses of the participants as Likert plots

Our findings confirm that IntelliGame encourages earlier implementation of functionality and, unlike the original study, also shows that IntelliGame enhances functionality by the end of the experiment.

4.5 RQ5: Does IntelliGame influence the developer experience?

According to the exit survey (Fig. 5a), the chosen target classes were appropriate, as participants from both groups reported having sufficient time to implement and test the class, and found it easy to understand and implement. Notably, fewer participants in the *treatment* group felt they had enough time to complete testing (46%) compared to the *control* group (59%), which is a significant difference ($p = 0.013$). This uncertainty among the *treatment* group is also reflected in their assessment of their test suite quality: only 27% of the *treatment* group believed they produced a good test suite,

```
test("Datetype - False", () => {
    const result = isPast(new Date(2024, 6, 13));
    expect(result).toBe(false);
});
```

Listing 1. Failing example test to check if the date is not in the past

```
test("Datetype - True", () => {
    const result = isFuture(new Date(2024, 7, 13));
    expect(result).toBe(true);
  });
```

Listing 2. Failing example test to check if the date is in the future

```
test("Add 10 days to 11/9 as string", ()=> {
    const test = new Date("9/11/2024");
    const result= addDays(test,10);
    expect(result.toLocaleDateString()).toBe('21/9/2024');
  })
```

Listing 3. Failing example test because of wrong asumption of date format

versus 38% in the *control* group, with this difference being nearly statistically significant ($p = 0.058$). A possible explanation is that IntelliGame users were more aware of gaps in their testing due to the achievement prompts and coverage reports, which the *control* group engaged with less often. This gave IntelliGame users a broader understanding of their test's quality and areas for improvement, whereas developers without IntelliGame are more likely to overestimate their testing efforts.

Among the participants of the *treatment* group who used the achievements (Fig. 5b), 40% felt that IntelliGame improved their programming habits, and about 50% were motivated by notifications, including both encouragement and progress updates. Most participants understood the achievements and how to attain them, and around half expressed interest in using IntelliGame in their regular work. Over 50% stated that IntelliGame motivated them to improve their testing practices, indicating that the plugin was effective in its goal.

Summary (*RQ 5*): IntelliGame effectively motivated the *treatment* group to improve testing practices and programming habits, with participants gaining greater awareness of test quality and areas for improvement, despite feeling slightly more time-constrained than the *control* group.

Our findings confirm that IntelliGame encourages participants to conduct more testing; however, unlike in the original study, the participants of the *treatment* group were less confident in their testing results compared to the *control* group.

4.6 RQ6: Do users of IntelliGame write high-quality tests?

After analyzing the participants' test suites, we found that the test smell ratio was consistently higher in the *treatment* group, with an average of 0.13 compared to the *control* group's 0.03, a statistically significant difference ($p = 0.013$, see Fig. 4c). One commonly observed test smell

```
1 console.log(isEqual(new Date(2024, 1, 22), new Date(2023, 1, 22)));
```

Listing 4. Test smells for using the console rather than an assert statement

is shown in listing 4, where a participant logged output to the console rather than using a Jest assertion. The significantly higher smell ratio in the *treatment* group suggests that participants may have been more focused on writing a high number of tests—perhaps to gain achievements—than on test quality. An obvious solution to counter this problem would be to enhance IntelliGame with a new category of test achievements related to test smells. However, upon examining the specific test smells, we identified a total of 108 instances in the participants' test code. Of these, 101 were console outputs, while the remaining seven involved conditional statements within tests. Further analysis revealed that many console statements appeared alongside assertions, suggesting they were used for testing rather than debugging but were not removed afterward. This indicates that the overall test quality may not be as poor as suggested by Smelly Test.

In RQ2 we noticed a substantial number of failing tests, when calculating mutation scores and therefore also investigated test robustness as an indicator of test quality. While the *treatment* group had a significantly higher mean number of failed tests (1.6) compared to the *control* group's 0.73 ($p = 0.037$, cf. Fig. 4d), to some extent this is influenced by the increased number of tests written by participants of the *treatment* group. The mean ratio of failing tests over tests written in total per participant is not significantly higher (*control*: 0.062, *treatment*: 0.114, $p = 0.095$).

One possible explanation for failing tests might be brittle tests due to the use of a date-time library as target, which presents specific robustness challenges. Date and time handling requires attention to details like future test execution, various time zones, and date formats, yet these factors were often overlooked. For example, one participant's test (listing 1) assumed a date would not be in the past—an assumption valid only during the experiment, leading to failures soon after. Another test in listing 2 assumed a date should always be in the future, which was also invalidated shortly after the experiment ended. Additionally, in listing 3, a participant's test expected output in `DD/M/YYYY` format instead of the system's `DD/MM/YYYY`. While such tests may have passed at the time the experiment was conducted, they failed during our later analysis. These cases suggest that test robustness with respect to date and time handling was not something the participants were particularly aware of.

To determine the effect of date-related problems, we re-ran the tests for each participant, this time adjusting the system time to match the commit time of the respective participant. Out of 214 initially failing tests, 61 passed with this adjustment, leaving 153 tests still failing due to other reasons. This reduction in failures is statistically significant ($p = 0.014$), and the ratios of failing tests are reduced to 0.042 (*control*) and 0.084 (*treatment*), still with no significant difference between the two groups ($p = 0.36$). Since brittle tests like in this scenario are not reported by common test smell detection tools, it may be worthwhile to investigate additional achievements for IntelliGame that explicitly reward robustness in tests.

To further analyze the remaining failures, we manually reviewed the still-failing tests and identified three primary causes: (1) incorrect expected values in assertions, (2) incorrect implementations, and (3) incorrect date output formats. Since most participants with failing tests had only one or two failures, it is likely that they simply ran out of time at the end of the session and were unable to fully fix their tests or implementations. However, we suspect one other possible reason for the slightly higher failure rate in the *treatment* group could be IntelliGame, as it rewards the number of tests and test executions rather than the number of *passing* tests. Future work could therefore enhance IntelliGame and its achievements to address this limitation in the future.

Table 1. Mean progress and p-values of all achievements available in IntelliGame. Detailed information about the achievements can be found in the original paper [36].

Achievement	Mean		p-value	Achievement	Mean		p-value
	Control	Treatment			Control	Treatment	
Test Executor	52.21	173.16	**0.031**	Check your classes	0.02	10.42	**<0.001**
The Tester	29.15	49.34	0.066	Check your branches	0.66	11.52	**0.022**
The Tester - Advanced	0	0	-	Class Reviewer - Lines	0.01	1.31	**<0.001**
Assert and Tested	19.67	32.2	0.69	Class Reviewer - Methods	0	0	-
Bug Finder	1.03	1.64	0.79	Class Reviewer - Branches	0	0	-
Safety First	16.67	17.19	0.63	The Debugger	1.25	2.09	0.087
Gotta Catch 'Em All	0.13	2.34	**<0.001**	Take some breaks	2.31	8.29	0.43
Line-by-Line	2.03	52.12	**<0.001**	Make Your Choice	0	0	-
Check your methods	0.47	8	**0.022**	Break the Line	2.31	8.29	0.43
Double check	17.87	19.12	0.73				

Summary (*RQ 6*): More test smells and lack of robustness indicate that the *treatment* group may focus on quantity over quality in their testing approach. Additional achievements in IntelliGame could be introduced to avoid these effects.

4.7 RQ7: What are the most important achievements?

Table 1 displays the achievements of IntelliGame developed for TypeScript, excluding those that are specific to Java. The table shows the average achievement actions for both the *treatment* and *control* groups, along with their p-values. An achievement action refers to any activity that contributes to progress toward the next achievement level. For instance, each time a participant uses the debugger, it counts towards the *Debugger* achievement, and each combination of a failing test followed by a modification to the source code contributes to the *Bug Finder* achievement.

The mean achievement levels were consistently higher in the *treatment* group across various categories. A large and significant difference is observable for test executions, and achievements related to test coverage were also notably more advanced in the *treatment* group, aligning with the observation that the *control* group did not run tests with coverage reports enabled. The three achievements with the highest number of actions in the *treatment* group were *Test Executor* (for executing tests), *The Tester* (for running test suites), and *Line-by-Line* (for covering lines with tests). This indicates that simpler actions during testing were also the most influential for achievement progress. These achievements likely encouraged participants to write tests, execute them, and increase line coverage, as demonstrated in Section 4.1 and Section 4.2.

Although there was significant progress on *Class Reviewer - Lines*, overall advancement in the *Class Reviewer* achievements was limited. While this might be another example of effects that would require longer periods of usage to manifest, an alternative conjecture is that this is influenced by the parameterization of the achievements. For example, the level thresholds set in our experiment may have prevented developers from progressing, thus limiting the potential of these achievements. Although we used the original IntelliGame thresholds, future work should re-examine these parameters to provide stronger empirical evidence for setting optimal values.

Indeed some achievements are designed for long-term engagement, which may not be observable within the timeframe of our experiment. For instance, achievements like *The Tester - Advanced* require running at least ten test suites with a minimum of 100 tests—an expectation that was not feasible within our experiment's duration. Thus, these types of achievements may be more appropriate for long-term application rather than short-term studies. This highlights the need for further replicability studies that examine the effects of gamification over extended periods.

The *Make Your Choice* achievement, which requires setting conditional breakpoints, was not utilized by any participants in the *treatment* group. While it is understandable that the *control* group

did not engage with this feature due to its obscurity, the lack of attempts by the *treatment* group is unexpected. More generally we observe low pursuit of debugging achievements, reflected in their low mean scores in Table 1. The *Bug Finder* achievement, which rewards participants for modifying source code following a failing test, highlights the difficulty developers face with debugging tasks. While the mean achievement level was higher in the *treatment* group, this difference was not statistically significant compared to the *control* group. This raises questions about the effectiveness of certain debugging achievements, especially those tied to specific types of breakpoints. Future iterations of the experiment might consider introducing new achievements that are easier to attain to better incentivize developers to engage in debugging behaviors.

Achievements such as *Assert and Tested* and *Double Check* were completed at similar rates by both groups, suggesting they may not have a strong impact in their current form. This calls for a re-evaluation of their design or reward structure to more effectively encourage developers to incorporate assertions in their tests in future implementations.

Finding the right balance between easily obtainable and more challenging achievements warrants further investigation. Given the lower completion rates for achievements like *The Tester - Advanced* and those related to debugging, introducing additional, intermediate achievements could help guide developers gradually toward more complex behaviors. Providing other forms of support, such as hints on how to achieve complex goals, could also be valuable for developers.

Summary (*RQ7*): While certain achievements in IntelliGame for TypeScript enhance user engagement and promote best practices in testing, the effectiveness of long-term achievements and debugging incentives varies, highlighting the need for further studies to explore their impact over extended periods and refine the achievement system.

5 Discussion

The original study found a significant improvement in branch coverage when users were exposed to gamification. In our experiment, although the treatment group produced slightly higher code coverage values, the differences between the *control* and *treatment* groups are not statistically significant. An explanation for this result might be the fact that inherently, the Date-Fns project does not have many branches by nature. Most of the functions indeed can be implemented with few lines of code and with a very limited number of conditional statements, therefore the variability of the measured branch coverage is limited. Future replicability studies may want to consider fewer, but more complex functions as experiment tasks, to allow for larger variations.

Besides coverage differences, in the original study test effectiveness was significantly improved for users creating the test suite with gamification, as quantified by the mutation score. In contrast, in this study we did not find a significant difference in this metric. Upon further examination of the mutants produced by the Stryker mutation tool, we noticed that for most of the functions to be implemented, the generated mutants were largely redundant (e.g., several mutants modifying the modulus operator with a multiplication). This redundancy can also be attributed to the inherent simplicity of certain functions, which, lacking complex logic, leads to the generation of redundant mutants that ultimately result in similar mutation scores.

Contrary to the original study, our participants in the gamified environment reported more time pressure and slightly less confidence in their test results. This might be influenced by the continuous feedback loop of achievement notifications and progress indicators, which may have raised awareness of test quality in general. In particular, if thresholds for achievement levels are set too high, then it would be conceivable that users would feel pressured to invest too much time into testing until the next level is reached. Consequently, it would be useful to revisit the thresholds currently set in IntelliGame, which have not yet been systematically explored and optimized.

The perceived time pressure may also have led some participants to prioritize quantity over quality, as seen in higher test smell ratios and increased failure rates in the *treatment* group. These findings suggest that, while gamification enhances testing engagement, there is potential to further refine achievement structures to foster both quality and quantity in test creation. An important oversight in the existing achievement system of IntelliGame revealed by our study is that failing tests count just like passing ones when IntelliGame counts the events that lead to awarding an achievement. However, adding failing tests should arguably only be rewarded if that test failure reveals an actual bug which is successively fixed. It is also conceivable to introduce further achievements related to test quality (e.g., awarding testers that remove test smells.)

Since the *treatment* group began testing earlier with the help of IntelliGame, they were able to identify and fix more bugs early on, whereas the *control* group initially neglected testing. Despite similar coverage and mutation scores, the higher number of tests in the *treatment* group suggests they spent more time detecting bugs early. This indicates that IntelliGame played a crucial role in improving implementation correctness rather than the difference being purely coincidental.

6 Conclusions

Our replication study confirms that gamification, through the IntelliGame plugin, significantly influences software testing behaviors in an IDE setting, supporting findings from the previous study and extending them with new insights. Gamification in the *treatment* group led to more frequent test creation, increased use of coverage tools, and engagement with debugging features, demonstrating IntelliGame's ability to encourage more rigorous testing practices. Notably, the *treatment* group also achieved a greater number of tests, indicating enhanced attention to finer details in test cases. These behaviors correlate with improved test suite quality metrics, higher achievement levels, and earlier functionality in the codebase.

Future research could explore adjustments to gamification frameworks within IDEs, focusing on balancing achievements that emphasize quality with those targeting quantity. Specifically, introducing achievements that reward robust test creation—such as those that prevent common test smells or require robust handling of time-sensitive data—could help improve the depth and reliability of testing practices. IntelliGame's incentive mechanism can be extended to balance test quality and quantity more effectively. Enhancing enforcement mechanisms for syntactic correctness, such as integrating a type-checking module or providing real-time feedback, could improve test input validity. Additionally, incorporating branch coverage visualization may help users construct more effective test cases. Furthermore, longer-term studies with larger sample sizes could help determine whether gamification's influence on testing behavior extends beyond the immediate experimental context and persists in professional development settings. Lastly, incorporating adaptive gamification, where the plugin tailors feedback based on individual tester progress, could further optimize user engagement and development outcomes.

7 Data Availability

The artifacts are available at https://doi.org/10.6084/m9.figshare.27443505

Acknowledgments

This study was carried out within the "EndGame - Improving End-to-End Testing of Web and Mobile Apps through Gamification" project (2022PCCMLF) – funded by European Union – Next Generation EU within the PRIN 2022 program (D.D.104 - 02/02/2022 Ministero dell'Università e della Ricerca). This manuscript reflects only the authors' views and opinions and the Ministry cannot be considered responsible for them.

References

[1] Carlos Futino Barreto and César França. 2021. Gamification in Software Engineering: A literature Review. In *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 105–108. https://doi.org/10.1109/CHASE52884.2021.00020

[2] Jonathan Bell, Swapneel Sheth, and Gail Kaiser. 2011. Secret ninja testing with HALO software engineering. In *Proceedings of the 4th International Workshop on Social Software Engineering* (Szeged, Hungary) *(SSE '11)*. Association for Computing Machinery, New York, NY, USA, 43–47. https://doi.org/10.1145/2024645.2024657

[3] Raquel Blanco, Manuel Trinidad, María José Suárez-Cabal, Alejandro Calderón, Mercedes Ruiz, and Javier Tuya. 2023. Can gamification help in software testing education? Findings from an empirical study. *Journal of Systems and Software* 200 (2023), 111647. https://doi.org/10.1016/j.jss.2023.111647

[4] Keith M Bower. 2003. When to use Fisher's exact test. In *American Society for Quality, Six Sigma Forum Magazine*, Vol. 2. American Society for Quality Milwaukee, WI, USA, 35–37.

[5] Filippo Cacciotto, Tommaso Fulcini, Riccardo Coppola, and Luca Ardito. 2021. A Metric Framework for the Gamification of Web and Mobile GUI Testing. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 126–129. https://doi.org/10.1109/ICSTW52544.2021.00032

[6] Peter J. Clarke, Debra L. Davis, Raymond Chang-Lau, and Tariq M. King. 2017. Impact of Using Tools in an Undergraduate Software Testing Course Supported by WReSTT. *ACM Trans. Comput. Educ.* 17, 4, Article 18 (Aug. 2017), 28 pages. https://doi.org/10.1145/3068324

[7] Riccardo Coppola, Tommaso Fulcini, Luca Ardito, Marco Torchiano, and Emil Alègroth. 2024. On Effectiveness and Efficiency of Gamified Exploratory GUI Testing. *IEEE Transactions on Software Engineering* 50, 2 (2024), 322–337. https://doi.org/10.1109/TSE.2023.3348036

[8] Igor Ernesto Ferreira Costa. and Sandro Ronaldo Bezerra Oliveira. 2019. A Systematic Strategy to Teaching of Exploratory Testing using Gamification. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC, SciTePress, 307–314. https://doi.org/10.5220/0007711603070314

[9] Gabriela Martins de Jesus, Fabiano Cutigi Ferrari, Daniel de Paula Porto, and Sandra Camargo Pinto Ferraz Fabbri. 2018. Gamification in Software Testing: A Characterization Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing* (SAO CARLOS, Brazil) *(SAST '18)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/3266003.3266007

[10] Gabriela Martins de Jesus, Leo Natan Paschoal, Fabiano Cutigi Ferrari, and Simone R. S. Souza. 2019. Is It Worth Using Gamification on Software Testing Education? An Experience Report. In *Proceedings of the XVIII Brazilian Symposium on Software Quality* (Fortaleza, Brazil) *(SBQS '19)*. Association for Computing Machinery, New York, NY, USA, 178–187. https://doi.org/10.1145/3364641.3364661

[11] Daniel de Paula Porto, Gabriela Martins de Jesus, Fabiano Cutigi Ferrari, and Sandra Camargo Pinto Ferraz Fabbri. 2021. Initiatives and challenges of using gamification in software engineering: A Systematic Mapping. *Journal of Systems and Software* 173 (2021), 110870. https://doi.org/10.1016/j.jss.2020.110870

[12] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. 2011. From Game Design Elements to Gamefulness: Defining "Gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments* (Tampere, Finland) *(MindTrek '11)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/2181037.2181040

[13] Association for Computing Machinery. 24-08-2020. Artifact Review and Badging Version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current. [Accessed 28-10-2024].

[14] Tommaso Fulcini and Luca Ardito. 2022. Gamified Exploratory GUI Testing of Web Applications: a Preliminary Evaluation. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 215–222. https://doi.org/10.1109/ICSTW55395.2022.00045

[15] Tommaso Fulcini, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2023. A Review on Tools, Mechanics, Benefits, and Challenges of Gamified Software Testing. *ACM Comput. Surv.* 55, 14s, Article 310 (July 2023), 37 pages. https://doi.org/10.1145/3582273

[16] Giacomo Garaccione, Tommaso Fulcini, Paolo Stefanut Bodnarescul, Riccardo Coppola, and Luca Ardito. 2024. Gamified GUI testing with Selenium in the IntelliJ IDE: A Prototype Plugin. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments* (Lisbon, Portugal) *(IDE '24)*. Association for Computing Machinery, New York, NY, USA, 76–80. https://doi.org/10.1145/3643796.3648459

[17] Giacomo Garaccione, Tommaso Fulcini, and Marco Torchiano. 2022. GERRY: a gamified browser tool for GUI testing. In *Proceedings of the 1st International Workshop on Gamification of Software Development, Verification, and Validation* (Singapore, Singapore) *(Gamify 2022)*. Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/3548771.3561408

[18] A. Jedlitschka and D. Pfahl. 2005. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.* 10 pp.–. https://doi.org/10.1109/ISESE.2005.1541818

[19] Natalia Juristo and Omar S. Gómez. 2012. *Replication of Software Engineering Experiments*. Springer Berlin Heidelberg, Berlin, Heidelberg, 60–88. https://doi.org/10.1007/978-3-642-25231-0_2

[20] Beáta Lőrincz, Bogdan Iudean, and Andreea Vescan. 2021. Experience report on teaching testing through gamification. In *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence* (Athens, Greece) *(EASEAI 2021)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/3472673.3473960

[21] Tobias Lorey, Stefan Mohacsi, Armin Beer, and Michael Felderer. 2022. Storm: A model for sustainably onboarding software testers. *arXiv preprint arXiv:2206.01020* (2022).

[22] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[23] Antonio Materazzo, Tommaso Fulcini, Riccardo Coppola, and Marco Torchiano. 2023. Survival of the Tested: Gamified Unit Testing Inspired by Battle Royale. In *2023 IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS)*. 1–7. https://doi.org/10.1109/GAS59301.2023.00008

[24] Edgy Paiva, Danielly Barbosa, Roberto Lima, and Adriano Albuquerque. 2010. Factors that Influence the Productivity of Software Developers in a Developer View. In *Innovations in Computing Sciences and Software Engineering*, Tarek Sobh and Khaled Elleithy (Eds.). Springer Netherlands, Dordrecht, 99–104.

[25] Yujian Fu P.E. and Peter J. Clarke. 2016. Gamification-Based Cyber-Enabled Learning Environment of Software Testing. In *2016 ASEE Annual Conference & Exposition*. ASEE Conferences, New Orleans, Louisiana. https://peer.asee.org/27000.

[26] Karl Pearson. 1895. VII. Note on regression and inheritance in the case of two parents. *proceedings of the royal society of London* 58, 347-352 (1895), 240–242.

[27] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, Robert Feldt, Antonio Filieri, Carlo Alberto Furia, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara A. Kitchenham, Romain Robbes, Daniel Méndez, Jefferson Molleri, Diomidis Spinellis, Miroslaw Staron, Klaas-Jan Stol, Damian A. Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, and Sira Vegas. 2020. ACM SIGSOFT Empirical Standards. *CoRR* abs/2010.03525 (2020). arXiv:2010.03525 https://arxiv.org/abs/2010.03525

[28] Muthu Ramachandran. 2003. Testing Software Components Using Boundary Value Analysis. In *29th EUROMICRO Conference 2003, New Waves in System Architecture, 3-5 September 2003, Belek-Antalya, Turkey*. IEEE Computer Society, 94–98. https://doi.org/10.1109/EURMIC.2003.1231572

[29] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 162–167. https://doi.org/10.1109/ICSTW.2016.43

[30] Martin Shepperd, Nemitari Ajienka, and Steve Counsell. 2018. The role and value of replication in empirical software engineering results. *Information and Software Technology* 99 (2018), 120–132. https://doi.org/10.1016/j.infsof.2018.01.006

[31] Swapneel Sheth, Jonathan Bell, and Gail Kaiser. 2015. A gameful approach to teaching software design and software testing. *Computer Games and Software Engineering* 9 (2015), 91.

[32] Klaas-Jan Stol, Mario Schaarschmidt, and Shelly Goldblit. 2021. Gamification in software engineering: the mediating role of developer engagement and job satisfaction. *Empirical Software Engineering* 27, 2 (30 Dec 2021), 35. https://doi.org/10.1007/s10664-021-10062-w

[33] Philipp Straubinger and Gordon Fraser. 2023. A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 80–90. https://doi.org/10.1109/ISSRE59848.2023.00075

[34] Philipp Straubinger and Gordon Fraser. 2024. Engaging Developers in Exploratory Unit Testing through Gamification. In *Proceedings of the 3rd ACM International Workshop on Gamification in Software Development, Verification, and Validation, Gamify 2024, Vienna, Austria, 17 September 2024*, Riccardo Coppola, Luca Ardito, Gordon Fraser, and Maurizio Leotta (Eds.). ACM, 2–9. https://doi.org/10.1145/3678869.3685683

[35] Philipp Straubinger and Gordon Fraser. 2024. Gamifying a Software Testing Course with Continuous Integration. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. 34–45.

[36] Philipp Straubinger and Gordon Fraser. 2024. Improving Testing Behavior by Gamifying IntelliJ. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 49, 13 pages. https://doi.org/10.1145/3597503.3623339

[37] Maneela Tuteja, Gaurav Dubey, et al. 2012. A research study on importance of testing and quality assurance in software development life cycle (SDLC) models. *International Journal of Soft Computing and Engineering (IJSCE)* 2, 3 (2012), 251–257.

[38] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.
[40] Zornitsa Yordanova. 2019. Educational Innovations and Gamification for Fostering Training and Testing in Software Implementation Projects. In *Software Business*, Sami Hyrynsalmi, Mari Suoranta, Anh Nguyen-Duc, Pasi Tyrväinen, and Pekka Abrahamsson (Eds.). Springer International Publishing, Cham, 293–305.

Received 2024-10-31; accepted 2025-03-31