Improving Testing Behavior by Gamifying IntelliJ

Philipp Straubinger
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

ABSTRACT

Testing is an important aspect of software development, but unfortunately, it is often neglected. While test quality analyses such as code coverage or mutation analysis inform developers about the quality of their tests, such reports are viewed only sporadically during continuous integration or code review, if they are considered at all, and their impact on the developers' testing behavior therefore tends to be negligible. To actually influence developer behavior, it may rather be necessary to motivate developers directly within their programming environment, while they are coding. We introduce *IntelliGame*, a gamified plugin for the popular IntelliJ Java Integrated Development Environment, which rewards developers for positive testing behavior using a multi-level achievement system: A total of 27 different achievements, each with incremental levels, provide affirming feedback when developers exhibit commendable testing behavior, and provide an incentive to further continue and improve this behavior. A controlled experiment with 49 participants given a Java programming task reveals substantial differences in the testing behavior triggered by *IntelliGame*: Incentivized developers write more tests, achieve higher coverage and mutation scores, run their tests more often, and achieve functionality earlier.

CCS CONCEPTS

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

KEYWORDS

Gamification, IDE, IntelliJ, Software Testing

ACM Reference Format:
Philipp Straubinger and Gordon Fraser. 2024. Improving Testing Behavior by Gamifying IntelliJ. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623339

1 INTRODUCTION

Although the importance of software testing is well established, unfortunately so is the fact that software is nevertheless generally inadequately tested [38, 48]. A common approach to ensure and improve test quality is to measure and visualize established metrics such as code coverage [60] or mutation analysis [33]. These analyses provide insights into how thoroughly a program is tested, and where there are holes in the test suite that could be improved with further tests. A common way these analyses are integrated into the developer workflow is by producing detailed reports during continuous integration or code review [3, 32]. While such reports are popular, there is little evidence [4, 20, 31, 41] that showing this information has any effects on test quality or developer behavior.

While it is safe to assume that software developers are nowadays well educated on the importance of testing and appropriate techniques to do so [29], it appears the problem is rather one of motivation: Despite the availability of coverage and mutation reports, testing is often not tightly integrated into the developer workflow [9] or simply not used. In order to improve test quality, it may therefore be necessary to fundamentally influence the testing behavior of developers directly while they are writing code. An approach that is commonly used in many domains to influence the behavior of humans is gamification, i.e., the integration of game elements into non-game contexts to incentivize humans to do things they are aware of but not sufficiently motivated to do. While the gamification of software testing has been investigated previously [7, 12, 42, 45, 47, 53], this has not been done directly within the development environment to the best of our knowledge.

In this paper, we therefore introduce an approach to incentivize developers to test more and better, by gamifying aspects of testing directly within their development environment, while coding. In particular, we create a catalog of 27 individual motivational aspects related to test quality, the use of test automation infrastructure, and the way testing activities are integrated into the workflow. Positive behavior according to these aspects, like measuring coverage when running tests, is rewarded by awarding developers multi-level *achievements*, which are designed incrementally such that initial positive reinforcement is achieved quickly, while there remains an incentive to further improve testing behavior to reach higher achievement levels. Achievements are therefore suited for supporting onboarding activities as well as for influencing the routines of more seasoned developers. We implemented this approach and the 27 achievements as a plugin for the popular IntelliJ Java Integrated Development Environment (IDE), and empirically study how it influences the workflow of software developers.

In detail, the contributions of this paper are as follows:

- We propose a gamification approach to incentivize and reward testing-related behavior while coding.
- We introduce the *IntelliGame* tool that implements this approach with 27 multi-level achievements integrated directly into the popular IntelliJ development environment.
- We empirically study the effects of *IntelliGame* using a controlled experiment with four different configurations and 49 participants tasked to implement a Java class.

We observe substantial improvements: Incentivized developers write more tests, achieve higher coverage and mutation scores, run their tests more often, and achieve functionality earlier.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623339

2 BACKGROUND

2.1 Developer Testing

Testing is an important aspect of software development, and developers are typically expected to write unit tests for the code they produce [51]. Although this is an accepted practice and developers often claim to spend substantial amounts of time on writing tests [9], it has been shown that they tend to overestimate how much testing they do [9]; indeed, there are many developers who do not test at all, and software projects without automated tests [9, 10].

In order to help testers improve their testing, a common approach lies in assessing test quality using metrics like code coverage or mutations analysis. Code coverage can be easily measured by executing a test suite and checking which parts of the code have been reached by the tests; there are various definitions of what are relevant 'parts', such as statements, branches, or execution paths. Developers receive feedback in terms of an overall percentage of how much of the program has been tested, or visualizations thereof [60]. Mutation analysis offers a stronger criterion by checking how many artificial faults can be detected by a test suite [33]. However, despite the existence of such means of support, software quality tends to be inadequate, leading to substantial costs during development and after releasing the software [38], suggesting that this feedback on test quality alone is not sufficient to motivate developers to test.

2.2 Developer Motivation

The suspected reasons for inadequate testing are manifold and include ubiquitous problems such as costs and time, but interestingly also developer motivation [35], which is generally seen as an essential factor for developer productivity [23–25]. The term *motivation* refers to either the inner willingness to participate in an activity for personal satisfaction (*intrinsic motivation*), or an external outcome that comes with or after completing a task, such as recognition [46] (*extrinsic motivation*). Although the terms *motivation* and *satisfaction* are often used interchangeably, they are different [23, 24]: Motivation needs to be present before the work, while satisfaction is a result of the work. However, they are connected because being satisfied with a task can motivate for the next one. When applied for a longer time, motivation can lead to *engagement*, which refers to commitment, hard work, and interest in one's current work and involves putting in more effort than required [18, 25, 54].

Prior work [15, 16] has investigated reasons why motivation is often missing when it comes to software testing. It has been found that software testing is often perceived as tedious, stressful and frustrating, such that surveyed students and professionals do not want to write tests in their work or follow the career path of a software tester [57, 58]. To motivate developers to test, they need variety in their work, creative tasks, recognition, and to gain new knowledge [15]. Unfortunately, there is a lack of applicable tools and techniques to provide these motivational aspects.

2.3 Gamification of Software Testing

Gamification aims to provide motivation for tasks that are normally not one's favorite job, and is commonly interpreted as the use of game design elements in non-game contexts [17]. Gamification can lead to more motivation as well as higher and better output of the work done by those working in gamified environments [14, 52].

The most commonly used gamification elements are points, badges, leaderboards, and awards [7, 12]. Many more elements can be used to gamify activities, depending on the context in which gamification is applied. Some elements may subsume other game design elements; for example, achievements can be a mixture of the game elements points, badges, levels, and awards, depending on how they are used or implemented [45]. An achievement can be gained by making progress in doing a specific task. If the task has been performed a certain number of times, a new level is reached and awarded with points or badges. To get to the next level, the task has to be performed more often than to reach the level before. This can be repeated multiple times until the last level is reached.

Gamification has been demonstrated to improve student motivation in learning software testing [13, 59]. There are several gamified learning environments [8, 19, 26, 27], mostly focusing on unit testing, utilizing elements such as points, leaderboards, achievements, and levels [12, 40]. There have also been attempts to gamify aspects of testing for professional developers, such as test-to-code traceability [42], acceptance testing [47], and unit testing [53]. The latter approach is particularly relevant as our objective is similar. However, prior work included gamification as part of continuous integration (CI) [53], where gamification elements depend on CI builds triggered by changes to the version control system. To the best of our knowledge [28], there have been no previous attempts to gamify software testing directly in the IDE during development.

2.4 IDE Support for Testing

Integrated Development Environments (IDEs) are a crucial part of every developer's daily work because they make it easier for them to write and test code. A code editor, compiler, debugger, UI builder and other tools are packed together into one single application with the ability to be extended by plugins so that the IDE can be modified to meet any requirements [30]. Most IDEs support the writing and execution of tests using different testing frameworks, which provide an easy way to write tests for a specific programming language, including execution engines and powerful debuggers. In addition, there are many tools available which support testing and quality of code in general, like test generation [5], code coverage [60], mutation testing [33] or detecting test smells [56]. Consequently, modern IDEs provide all the ingredients necessary for effective testing, yet so far they are lacking incentives for developers to do so.

3 GAMIFICATION PLUGIN FOR INTELLIJ

Our approach focuses on integrating gamification elements directly into IDEs, and this is based on several design decisions:

- **Design Decision 1:** Our goal is to encourage developers to write and use tests. As intrinsic motivation cannot be influenced by external factors, we target extrinsic motivation: Developers should experience satisfaction and a boost in motivation for testing, for which we use *gamification*.
- **Design Decision 2:** We recognize that many test interactions occur locally on developers' machines, and not just following CI builds [53]. We therefore aim to motivate *directly in IDEs*. Specifically, we developed a plugin for IntelliJ

Table 1: Overview of implemented achievements with their descriptions and level boundaries

Achievement	Description	Bronze	Silver	Gold	Platinum
Testing					
Test Executor	Execute tests	3	100	1,000	10,000
The Tester	Run test suites	3	100	1,000	10,000
The Tester — Advanced	Run test suites X times containing at least Y tests	X: 10 Y: 100	X: 50 Y: 500	X: 100 Y: 1000	X: 250 Y: 3,000
Assert and Tested	Trigger AssertionErrors	3	10	100	1,000
Bug Finder	Previously failed test passes again after source code change	3	10	100	1,000
Test Fixer	Previously failed test passes again after test code change	3	10	100	1,000
Safety First	Write tests	10	100	1,000	10,000
Coverage					
Gotta Catch 'Em All	Run test suites with coverage	3	10	100	1,000
Line-by-line	Cover lines with your tests	100	1,000	10,000	100,000
Check your methods	Cover methods with your tests	10	100	1,000	10,000
Check your classes	Cover classes with your tests	10	100	1,000	10,000
Check your branches	Cover branches with your tests	10	100	1,000	10,000
Class Reviewer - Lines	Cover X classes with at least Y lines by Z% coverage	X: 5 Y: 5 Z: 70	X: 20 Y: 25 Z: 80	X: 75 Y: 250 Z: 85	X: 250 Y: 500 Z: 90
Class Reviewer - Methods	Cover X classes with at least Y methods by Z% coverage	X: 10 Y: 3 Z: 60	X: 50 Y: 8 Z: 80	X: 250 Y: 15 Z: 85	X: 500 Y: 25 Z: 90
Class Reviewer - Branches	Cover X classes with at least Y branches by Z% coverage	X: 5 Y: 15 Z: 75	X: 20 Y: 50 Z: 80	X: 75 Y: 250 Z: 85	X: 250 Y: 500 Z: 90
Debugging					
The Debugger	Run the code in debug mode	3	10	100	1,000
Take some breaks	Set breakpoints	10	100	1,000	10,000
Make Your Choice	Set conditional breakpoints	3	10	100	1,000
On the Watch	Set field watchpoints	3	10	100	1,000
Break the Line	Set line breakpoints	3	10	100	1,000
Break the Method	Set method breakpoints	3	10	100	1,000
Console is the new Debug Mode	Use System.out.println instead of debugger or logger	3	10	100	1,000
Test Refactoring					
Shine in new splendor	Change source code between two ensuing passing test runs	5	50	500	2,500
The Eponym	Rename test method names	10	100	1,000	10,000
The Method Extractor	Extract code from tests into a separate method	10	100	1,000	10,000
The Method Inliner	Inline methods into tests	10	100	1,000	10,000
Double check	Add new assertions to already passing tests	3	10	100	1,000

IDEA from JetBrains[1], which is the most widely used IDE in the JVM community [1]. However, the approach itself is not dependent on the specific IDE used.

- **Design Decision 3:** Rather than suggesting specific tasks that may disrupt their current workflow [53], we want to allow developers to choose the testing tasks they find relevant. In addition, we believe that feedback and rewards should be provided quickly, visibly, and immediately after an action to incentivize developers and give them clear goals. We therefore choose the gamification element of *achievements* to provide encouraging feedback immediately when we detect that what a developer did is commendable.
- **Design Decision 4:** We target short-term interventions, for example, to onboard new developers, or to influence the testing behavior of established developers but nevertheless need to ensure users remain engaged with the gamification as long as they make progress. We therefore provide not only variation in terms of simpler as well as more challenging goals for the achievements, but also different *levels* of achievements and *progress* in between levels.
- **Design Decision 5:** We want to nudge developers to participate in testing if they do not do this on their own, and therefore use *notifications* to inform developers of their progress towards available achievements and provide a visual representation of the achievements where the progress is shown.

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

To implement *IntelliGame* with respect to these design decisions, we roughly followed an iterative design research approach: Starting with our own observations, a literature review on gamification of testing as well as developer testing, and a technical inspection of data accessible through IntelliJ's APIs, we defined an initial set of test achievements and implemented a first prototype of *IntelliGame*. Feedback on this prototype and its achievements was collected from local researchers and students, and informed refinements as well as further achievements. We then proceeded to define the study task (Section 4.1.1) and procedure (Section 4.1.3), and conducted a pilot study, which informed further improvements, for example by adjusting parameters and thresholds of achievements. After a final round of refinements, we arrived at the version of *IntelliGame* also used in the evaluation study presented in this paper.

3.1 Testing Achievements

When designing the achievements, we tried to adhere to the general criteria [15] that good achievements (1) set clear goals, (2) provide visible progress, (3) offer a variety of tasks, (4) provide recognition, and (5) allow developers to gain new knowledge. Each test achievement therefore consists of five elements:

(1) A catchy, memorable title.
(2) A description of the aspect that is rewarded.
(3) A progress value that represents the current status with respect to the aspect that the achievement captures.
(4) An underlying mechanism that monitors user actions and updates the progress value accordingly.

(5) Level boundaries, which define when users are awarded new levels. For each achievement, there are bronze, silver, gold, and platinum levels. For example, a developer needs to execute three tests to get the bronze level of *Test Executor*, 100 for silver, 1,000 for gold and 10,000 for platinum.

The monitoring mechanism for an achievement subscribes to user events triggered in the IDE. The simplest type of achievement simply counts the number of occurrences of an individual type of action, such as how often a user executed a test suite, how often assertions were triggered, or how often breakpoints were set. The progress value can also reflect variables dependent on occurring events, such as the number of individual tests executed when the user runs a test suite, or coverage levels achieved during such executions. Achievements define level boundaries on doing tasks that allows progress; for example, progress may be achieved if a certain coverage threshold has been reached. Finally, achievements may also monitor not just individual events, but more complex sequences of actions that together witness a certain testing-related behavior. For example, repairing a broken test consists of multiple actions: First the user needs to run a test suite in which a test fails, then the test code is edited, and at last the test is re-run successfully, all while the code itself remains unchanged.

We derived objectives that represent good testing behavior based on common testing curricula [6, 34] and best practices [37, 50], such as: (1) ensuring that tests are written and executed after changes, in order to identify bugs in both test and production code [6, 50]; (2) using metrics such as code coverage to assess the quality of the test suite and identify test gaps [6, 37]; (3) using tests to support debugging [6]; (4) constantly refactoring, adjusting, and simplifying both test and source code to ensure high maintainability [37]. Finally, the design of achievements is also constrained by the information about the developer actions that can be extracted from IntelliJ (Section 3.3). Consequently, we distinguish between types of achievements based on the aspect of testing they refer to:

- **Testing Achievements**: Developers should write and run tests, and this category aims to incentivize developers to do this more often. Achievements also cover aspects of interacting properly with the tests, such as triggering assertions, fixing failing tests, or fixing bugs detected by the tests.
- **Coverage Achievements**: Achievements in this category aim to incentivize developers to evaluate the quality of their test suite by using coverage information more often, and by improving the coverage achieved by their tests.
- **Debugging Achievements**: If the tests reveal problems, they should be used as well as possible to help identify the cause. Achievements in this category aim to incentivize the use of the IDE's debugging features in conjunction with tests, e.g., running tests in debug mode or with breakpoints set. This implicitly also rewards writing tests, because having good tests is a prerequisite for effective debugging.
- **Test Refactoring Achievements**: Achievements in this category aim to incentivize developers to improve existing test code by applying test refactorings, such as extracting redundant code into helper methods.

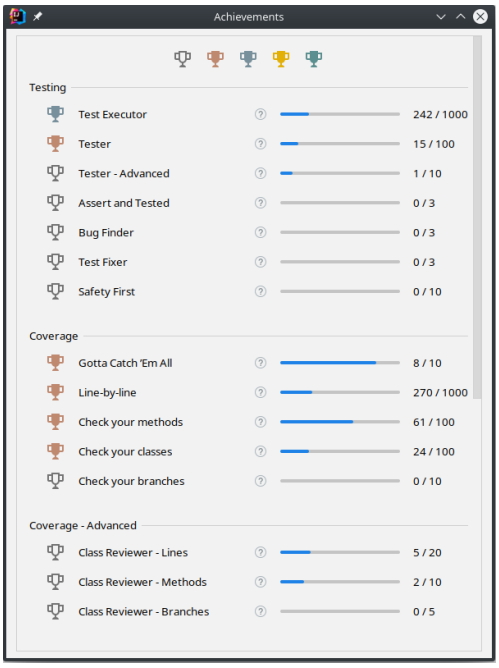


Figure 1: IntelliJ window showing achievements and progress

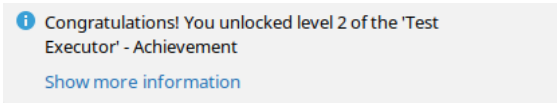


Figure 2: Example notification showing that level 2 of the achievement 'Test Executor' has been reached

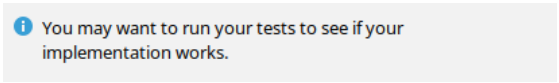


Figure 3: Example notification encouraging to execute tests

Based on these four categories, we define 27 achievements, summarized in Table 1. The level boundaries for these achievements were determined experimentally during a pilot study (Section 4.1.1).

3.2 User Interface

The achievement overview shown in Fig. 1, accessible in the Tools tab of IntelliJ, provides information on available achievements and their current progress. Each achievement is depicted with a trophy, which represents the level, a progress bar representing the current progress with respect to the next level, and a description of what progress is needed to reach the next level, which can be shown by hovering over the question mark. Whenever a new level has been reached, the progress bar and the information about the next target are updated. The progress value itself is not reset after reaching a new level, such that users always can see their overall progress. After reaching the final (platinum) level for an achievement, the

progress value may continue increasing, but will not lead to updates of the progress bar because the last level has already been reached. At this point, we expect the developers to have adopted that practice and therefore no longer require *IntelliGame*. The current progress is not project-dependent, new levels can be achieved in all opened projects. The progress can also be reset in the Help tab in IntelliJ.

The plugin also shows notifications: First, whenever a new level of an achievement has been reached, the user is notified as depicted in Fig. 2; the notification also provides a clickable link to the achievement overview. Second, a notification is shown when the developer makes progress in one of the achievements. To avoid excessive notifications, these are only shown every 25 % of progress. Third, there are notifications aimed to incentivize developers to improve progress for reaching the next level and solving new achievements (Fig. 3). Encouraging notifications are shown right after installation, and then after a configurable duration without any progress (which we set to five minutes for experiments, and 30 minutes in general.)

3.3 Implementation

IntelliJ is designed to be extensible and was originally implemented to support the extension with plugins. There is rich documentation about this topic on the website of JetBrains[2], and IntelliJ itself is open source[3]. In addition to extending given functionality in IntelliJ, many user actions and execution results can be queried, monitored, and analyzed. This is implemented using the publisher-subscriber-pattern [44], where IntelliJ provides several publishers, to which a plugin can subscribe to then be notified about changes or user interactions, such as test runs and their results, changes in the project's structure and files, coverage after a test run with coverage, and information on debugger and breakpoints usage.

To implement achievements of the category Test Refactoring we use RefactoringMiner [55], which compares edited files before and after a change and derives information about applied refactorings. In particular, *Shine in new splendor* is implemented by checking whether RefactoringMiner reports any refactorings for a change (that follows a test execution and is succeeded by another one). We use the *renaming* refactoring to identify *The Eponym*, the *extracting* refactoring for *The Method Extractor*, and the *inlining* refactoring for *The Method Inliner*. *Double check* is implemented by directly comparing successive versions of test code.

4 EVALUATION

In order to evaluate whether the gamified development environment influences the developers' behavior, we conducted a controlled experiment and aim to answer the following research questions:

- **RQ 1**: Does *IntelliGame* influence testing behavior?
- **RQ 2**: Does *IntelliGame* influence resulting test suites?
- **RQ 3**: Do achievement levels reflect differences in test suites and activities?
- **RQ 4**: Does *IntelliGame* influence the functionality of resulting code?
- **RQ 5**: Does *IntelliGame* influence the developer experience?

[2]https://plugins.jetbrains.com/docs/intellij/welcome.html
[3]https://github.com/JetBrains/intellij-community

4.1 Experiment Setup

The controlled experiment, consisting of a programming task, was carried out at the University of Passau in multiple sessions in June and July 2022 as well as February 2023.

4.1.1 Experiment Task. To improve ecological validity of the experiment, we aimed to base the programming task on code taken from a real software project. In addition, the task should be challenging, implementable and testable in 60 minutes, have a good specification for clear and easy understanding as well as no complex structures and dependencies. We found suitable candidate tasks in the work of Rojas et al. [49], since their selected classes from Apache Commons meet these requirements. From this data set, we used the FixedOrderComparator class[4], because every developer should be familiar with the general concept of a comparator, which reduces the time needed for understanding the task. In addition, the class and its functionality are easy to understand and test, and Rojas et al. revised and extended the JavaDoc specification.

We conducted a pilot study with five participants, the data of which is not included in the final experiment and our analysis. The participants of the pilot study were required to implement and test the whole Java class using JUnit tests from scratch, half of them with and without *IntelliGame*. After the study, it became clear that 60 minutes would not be sufficient time to complete the implementation and write adequate tests. Therefore, we simplified the task by providing a scaffolding consisting of the class with the contents of the two methods addAsEquals() and compare() removed except for the signature and JavaDoc comments.

Furthermore, we used the data gathered during the pilot study to adjust the level boundaries of the achievements to values realistically usable in practice: We extracted the numbers of actions performed by the participants and set the level boundaries so that it would take a few weeks to reach the platinum level when working continuously. Not every boundary could be extracted during the pilot study, because the participants did not work on every achievement, and we estimated appropriate values for the remaining ones.

In addition to the FixedOrderComparator class, a Main class was given to the participants, which makes it easier to apply manual testing, i.e., testing within the main method. We also prepared the folder structure including a test folder, such that there would be no configuration effort for participants to add tests, if they decided to add a test class (e.g., FixedOrderComparatorTest).

4.1.2 Participant Selection. To recruit participants we created an eligibility survey consisting of basic demographic questions, questions about programming experience in Java and different testing tools for Java, as well as five technical questions about JUnit to assess the testing knowledge. Each of the single-choice technical questions is based on a small code example with four answer options[5]. We advertised the survey to all (former) computer science students who had taken one of our software engineering-related courses within the last year. Thus, all participants had previously passed courses with advanced Java-based programming assignments and mandatory coverage-based unit testing, which increases

[4]https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/comparators/FixedOrderComparator.html
[5] Detailed information included in the artifacts

confidence that they are familiar with Java, JUnit, and debugger and coverage tools.

We received 62 responses to the survey. As minimum qualification we required at least three out of the five technical questions answered correctly as evidence of expertise, which 49 respondents satisfied and were therefore chosen as our participants; 86 % of the participants were students, while the rest were graduates now working in industry (8 %) or university (6 %). The age ranges from 20 to 34, with 43 male and six female participants. Most participants (76 %) claimed to have more than three years of experience with Java, and more than one year with JUnit (59 %). Each participant received a 15 € Amazon voucher as an expense allowance.

4.1.3 Experiment Procedure. The experiment was conducted in multiple sessions in person in the computer lab of the University of Passau. Each experiment session started with a 15-minute introduction explaining the experiment procedure and the programming task based on examples and documentation of how the class should behave. We furthermore provided each participant with a printed specification of the two methods to implement.

We divided participants into four groups:

- **Control group**: Participants have to solve the task without the plugin.
- **Treatment group**: Participants have to solve the task with the plugin enabled, without being required to use it.
- **Notifications group**: Participants have to solve the task with motivational notifications shown during the development process but without visible achievements.
- **Maximizing group**: Participants have to solve the task with the plugin, and they are explicitly asked to solve as many achievements as possible.

The *control* and *treatment* groups are our main analysis focus since comparing these two groups reveals the difference in developing with and without gamification. Since the plugin shows notifications together with achievements, it is unclear whether any observed effects result from the achievements or are simply triggered by notifications. To discern these effects, we evaluate the notifications as a stand-alone feature with the *notifications* group, which receives only generic notifications (e.g., Fig. 3) but no notifications about achievements. A further concern may be whether any observed differences are a result of participants focusing on the achievements rather than programming, for example, because they incorrectly assume that is what we expect them to do. To distinguish such effects from participants focusing on development, the *maximizing* group consists of participants explicitly tasked to solve as many achievements as possible.

We organized six experiment sessions in two blocks, wherein the first one (4 sessions) the participants were assigned to either the *control* or *treatment* groups, and in the second block (2 sessions) to the *notifications* or *maximizing* groups. The participants were alternatingly assigned to the groups based on the sequence in which they entered the room. We assigned more participants to the *control* and *treatment* groups since these represent our main focus. Of the 49 participants, 17 were assigned to the *treatment* group, 16 to the *control* group and eight each to the *notifications* and *maximizing* groups. In addition to general explanations[5], participants in the *treatment* and *maximizing* groups were shown the plugin individually. The goal of the study was not revealed until after the study to avoid biasing behavior.

Following the introductory explanations, participants had 60 minutes to implement the Java programming task in an IntelliJ environment. The task was (1) to implement the methods addAsEquals() and compare() based on the explanations and documentation given[5], and (2) to ensure the correctness of the methods' functionality. We did not explicitly instruct participants to write unit tests, but discussed possibilities like unit tests or manual testing with the main method. Throughout the experiment, a custom event logger saved the states of the achievements after each user interaction. A custom script committed and pushed the current implementation and log files to a Git repository once per minute.

After finishing the 60-minute implementation phase, participants completed an exit survey online, consisting of general questions about the implementation and testing of the class. A second page, shown only to the *treatment* and *maximizing* groups, asked about experiences with the plugin. Answer options were based on a five-point Likert scale, and the questions can be viewed in Section 4.6. An optional free-text question allowed to provide further comments.

4.1.4 Experiment Analysis. The analysis of the experiment is based on comparing results between the four groups. To determine the significance for any of the measurements between those groups, the exact Wilcoxon-Mann-Whitney test [39] was used to calculate the p-values with $\alpha = 0.05$. When visualizing trends we show the 84.6 % confidence intervals for calculating the means [22]; if the intervals overlap there is no statistically significant difference [2] (which is equivalent to checking significance using an exact Wilcoxon-Mann-Whitney test with $\alpha = 0.05$).

RQ 1: Does IntelliGame *influence testing behavior?* We compare the testing behavior in terms of (1) test creation, (2) test executions, (3) coverage measurement, and (4) use of the debug mode to run tests. Since the programming task involves an API rather than a program with a user interface, it is always necessary to write some test driver code to exercise the implemented code. The desirable approach to do so is to put this code into JUnit tests; a less desirable approach, akin to manually testing a program, is to temporarily place this code into the Main class. Even though this code is typically deleted again afterward, we can consider the entire history of the code and thus determine how often manual testing was applied. We consider all methods annotated using the JUnit @Test annotation as tests, regardless of their content. The number of test executions, with and without coverage or debugging, is collected by our plugin, which is running even if the UI is not visible (for the *control* and *notifications* groups). We also consider the number of tests, test executions with and without coverage, and debug uses over time based on the commit history. In addition, we calculate significant differences in testing between the groups with the exact Fisher test [11] with $\alpha = 0.05$.

RQ 2: Does IntelliGame *influence resulting test suites?* To answer this question, we compare the four experimental groups in terms of (1) the number of tests, (2) code coverage, and (3) mutation scores of the final test suites. We again take only methods with the @Test annotation into account as tests. We measured the code coverage

of the implementation of the participants with the help of JaCoCo[6]. We decided to not use the internal code coverage report of IntelliJ because JaCoCo is more widely used and can be automated. Failed tests were excluded from coverage measurement. Only the two methods to be implemented as well as added helper methods were included in the coverage measurements. We used both line and branch coverage to get a complete overview of the covered code. To calculate mutation scores we used the PIT[7] mutation testing system for Java. PIT was integrated into each project of the participants as a Maven plugin and configured to generate mutants for the methods addAsEquals() and compare() only. As constructors could not be excluded in the mutation analysis, we manually removed these mutants for calculating the mutation scores. Since PIT expects test suites without failing tests, all failing tests were excluded. In all cases, these are tests the participants were actively working on but did not have the time to complete when the experiment ended.

RQ 3: Do achievement levels reflect differences in test suites and activities? IntelliGame logs the current progress of each achievement as well as the reached levels after each user interaction during the experiment. For each participant, we extract the total number of times a new level was reached on any of the achievements in the end and during the study to extract the levels reached every minute during the study. We then compare this number between the four groups and measure the Pearson rank correlation [43] with line coverage, branch coverage, mutation score, and number of tests.

RQ 4: Does IntelliGame *influence the functionality of resulting code?* To answer this question, we used the golden test suite from Rojas et al. [49], which consists of six test cases. We executed these tests against the final version and all intermediate versions based on the commit history of each participant's code, and compare the numbers of passing tests between groups.

RQ 5: Does IntelliGame *influence the developer experience?* This question is answered by comparing the answers to the exit survey. For better visualization, the data is presented in stacked bar charts including the questions and percentages.

4.1.5 Threats to Validity. There are potential *threats to external validity* of our study. We only focused on one specific implementation task, which means that our findings may not be applicable to other tasks. However, since unit testing, debugging, and refactoring are fundamental concepts used in various programming languages and applications, we believe that our approach can be generalized to a broader context. Additionally, our experiment was conducted with participants from a specific university only, and results may differ if the experiment was repeated with participants from other universities or from industry. Furthermore, our evaluation of *IntelliGame* was limited to a short-term experiment, which may restrict the generalizability of our findings to long-term usage scenarios. However, *IntelliGame* is designed to be adaptable and can be easily extended for long-term studies.

[6] https://www.jacoco.org/jacoco/
[7] https://pitest.org/

Threats to internal validity may arise from errors in our data collection infrastructure, plugin, or experiment procedure. We thoroughly tested all software and validated it in a pilot study. Furthermore, the groups could be imbalanced with respect to demographics or programming skills. However, there were no significant differences in age, gender, and occupation between the groups according to the exact Wilcoxon-Mann-Whitney test [39] with $\alpha = 0.05$. In terms of expertise, there were no significant differences except that the *treatment* group claimed more self-declared expertise in Java ($p = 0.011$) and JUnit ($p = 0.017$) compared to the *maximizing* group, which, however, does not negatively affect any of our conclusions. Our qualification questions may not be precise enough to fully capture knowledge and experience. To mitigate this potential issue, we selected participants who had been enrolled in a course that covered unit testing. Additionally, we assessed their understanding of testing with JUnit by asking five questions, and out of 245 responses (five questions for each of the 49 participants) we only identified 16 incorrect answers. A perfectly balanced distribution of participants would likely require a closer assessment of skills, which would inhibit recruitment.

Our recruitment process may have introduced a selection-bias [36] as all participants voluntarily chose to take part. A higher number of participants who have a natural inclination or motivation towards testing may skew results in terms of more tests, achieved levels, or coverage compared to what an average population might achieve. To mitigate this threat, we only advertised the experiment as programming task without explicitly mentioning the aspect of testing or gamification, thus hoping for a more representative sample of students with interest in programming. To avoid that prior knowledge of the FixedOrderComparator class influences results or that the code would be found through a web search, all references to the originating software project and authors were removed. To minimize external influences we conducted the experiment in a controlled environment at the University of Passau.

There are potential *threats to construct validity* of our study. One concern is that the golden test suite may not encompass all possible edge cases. The golden test suite achieves 90 % line coverage and an 86 % mutation score in both methods; while it does not fully cover statements related to input validation, it does test everything related to the main functionality. Another potential threat is the setup of our experiment, which took place in a controlled lab session. Like all controlled environments this may not accurately represent the experiences and behaviors of all developers in real-world scenarios.

4.2 RQ 1: Does *IntelliGame* influence testing behavior?

Test Creation. IntelliGame intends to incentivize writing unit tests, and appears to achieve this: There was only one participant in the *treatment* group who did not write JUnit tests at all, but six participants of the *control* group which is significant ($p = 0.039$) according to the exact Fisher test [11]. In contrast, participants of the *control* group used the main method for manual testing more frequently: Only 2/17 participants in the *treatment* group used the main method for testing at some point, compared to 7/16 of the *control* group. Notifications show some effect, but less than the achievements: 2/8 participants wrote no unit tests, and 2/8

used main testing which can be seen as an indication that the participants of the *notifications* group are slightly more motivated than the *control* group, but less than the *treatment* group. Focusing on achievements seems to lead to the most motivation [25] to test: All participants of the *maximizing* group wrote unit tests, and only one of them used main testing, which suggests that achievements make developers test more than needed to complete the study task.

Figure 4a shows the JUnit tests written over time: Both the *treatment* and *control* groups start early to write tests; the first participant of the *control* group wrote a test after a minute, and for the *treatment* group after two minutes. From 22 minutes onwards the number of tests in the *treatment* group increases much faster than for the other groups (except *maximizing*), and from minute 30 on the improvement over the *control* group is significant (the confidence intervals do not overlap.) The *notifications* group initially behaved similarly to the *treatment* group, but stagnated towards the end and stayed between both the *control* and *treatment* groups, confirming that notifications contribute to, but do not fully explain, the improvements. As expected, the *maximizing* group started sooner and wrote significantly more tests than any other group between minutes 31 and 50, at which point the *treatment* group caught up.

Test Executions. Figure 5a shows the number of test executions, with a mean of 22.8 for the *treatment* group and 10.5 for the control group. The *notifications* ($p < 0.01$), *treatment* ($p < 0.01$) and *maximizing* ($p < 0.01$) groups all executed significantly more tests than the *control* group. In the *maximizing* group, two values with more than 1.5 million and 250,000 test executions (achieved using IntelliJ's feature of repeating test executions a configurable number of times) were omitted from the graph to make it more readable. These two data points demonstrate the possible exploitation of *IntelliGame* when the participants try to get as many achievements as possible, which may distract the developers from their actual goal. Consequently, it seems important not to make the use of *IntelliGame* mandatory for developers, but rather to let them decide on their own how to use achievements as was done in the *treatment* group. In addition, Fig. 4b illustrates the test executions over time together with the 84.6 % confidence intervals. A significant difference between the *control* and *treatment* groups can be observed starting from minute 27. The *notifications* and *control* groups behave similarly to each other, while the *maximizing* group shows a significant difference to the *control* group in the end. Overall, participants with *IntelliGame* run their tests more often.

Coverage Measurement. Figure 5b compares the number of times tests were executed with coverage collection activated: Only one participant in the *control* group used this functionality of IntelliJ 10 times, resulting in an average number of coverage executions of 0.6 overall participants. On the other hand, the *treatment* group used the coverage report with a mean of 4.7 executions per participant. Figure 4c shows the coverage executions over time, demonstrating that the first participant of the *treatment* group used the coverage report after 13 minutes, whereas in the *control* group only after 35 minutes. This also means that the difference is significant from the beginning on ($p < 0.01$), when the first participant executed their tests with coverage collection activated. The *notifications* group behaves similarly to the *control* group, even though there is a dedicated notification that suggests running tests with coverage. However, this is only shown if tests are run without coverage. The *maximizing* group as expected measures coverage the most during the last third except the very end. Overall, the participants of the *treatment* and *maximizing* groups seem to care more about coverage.

Debug Executions. Figure 5c shows how often tests were executed using the debug mode: On average participants of the *treatment* group used debug executions 2.6 times, and of the *control* group 1.8 times, although the difference is not significant ($p = 0.82$). This can also be seen in Fig. 4d, since the confidence intervals of all groups except for the *maximizing* group overlap. We can see a significant difference halfway through the experiment, but the overall difference is not significant ($p = 0.074$). We assume that debugging will be more common during maintenance and bug fixing, whereas our experiment task was to write new code. Furthermore, since the class under test is quite simple, the participants may not have had to use the debug mode to locate bugs found by failed tests.

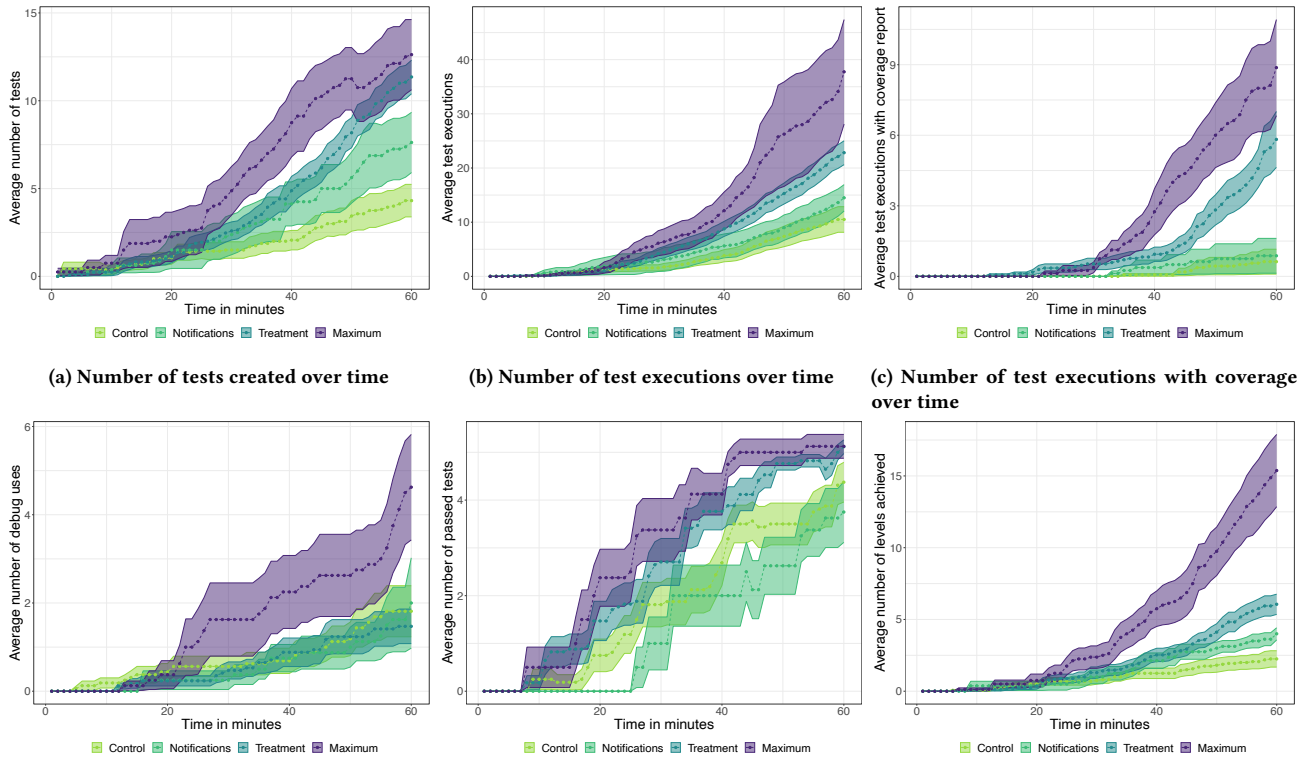
Summary (*RQ 1*): *IntelliGame* influences the testing behavior of the participants: They use JUnit tests rather than main testing, create more tests, run their tests significantly more often, and use coverage reports more frequently.

4.3 RQ 2: Does *IntelliGame* influence resulting test suites?

Number of tests. Figure 5d compares the resulting test suites in terms of the number of JUnit tests they consist of. The participants of the *treatment* group wrote a mean of 11.5 tests, while the *control* group wrote 4.3 tests and the *maximizing* group 12.5. According to the exact Wilcoxon-Mann-Whitney test, the *treatment* ($p < 0.01$) and *maximizing* ($p = 0.036$) groups wrote significantly more tests than the *control* group. One can also see that the *notifications* group has more tests in their test suites (mean 7.38) than the *control* group (mean 4.31), which again confirms that notifications influence developers to write more tests.

Code coverage. Figure 5e compares the line coverage between the four groups, showing that the *treatment* ($p < 0.01$) and *maximizing* ($p < 0.01$) groups have significantly higher line coverage than the *control* group. On average, the *treatment* and *control* groups achieve 89 % and 48 % line coverage, respectively. In the *treatment* group, there is one outlier without any coverage; this participant did not write any unit tests but only tested manually. When asked about this peculiar behavior, the participant stated to have rushed to complete and leave the study as quickly as possible. Results are similar for branch coverage (Fig. 5f), which also shows a significant difference between the control, *treatment* ($p < 0.01$) and *maximizing* ($p < 0.01$) groups, with a mean branch coverage of 85 % and 41 % for the *treatment* and *control* groups. The *notifications* group has no significantly different line ($p = 0.44$) or branch coverage ($p = 0.64$) compared to the *control* group, which is plausible since notifications only suggest measuring coverage, but not maximizing it. Overall, the significantly higher coverage for both the *treatment* and *maximizing* groups confirms the influence of gamification on test suite quality as measured with code coverage.

Mutation score. The mean mutation score (Fig. 5g) of the *treatment* group is 84 %, while the *control* group achieves a mean score



(a) Number of tests created over time

(b) Number of test executions over time

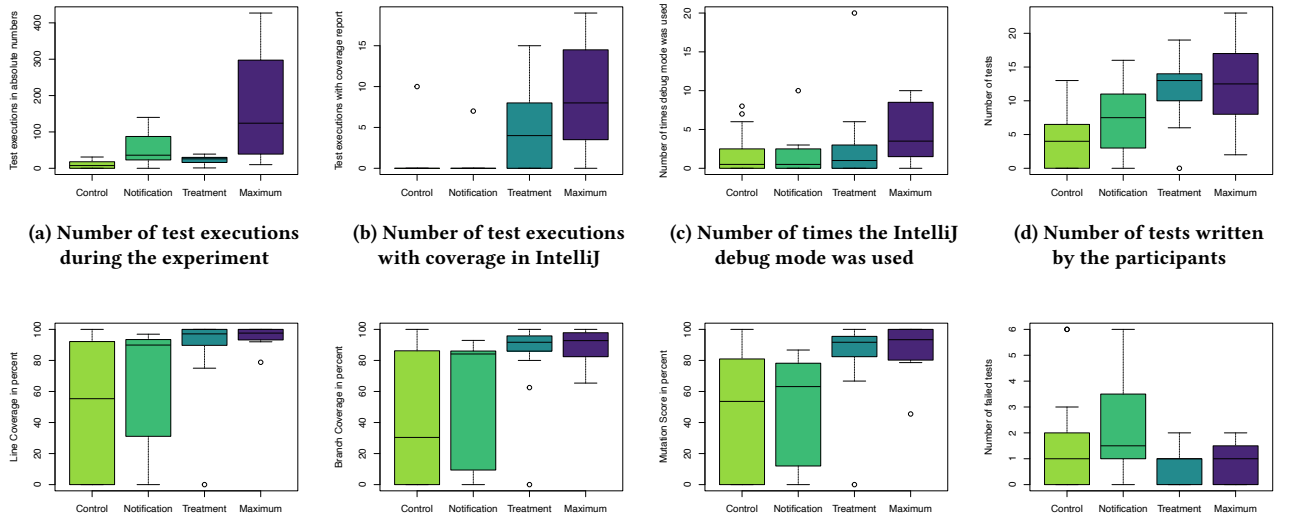
(c) Number of test executions with coverage over time

(d) Number of debug uses over time

(e) Number of passed tests over time

(f) Number of levels over time

Figure 4: Differences between *control*, *notifications*, *treatment* and *maximizing* groups over time (see Section 4.1.3)



(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

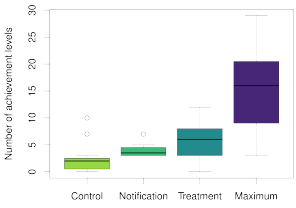
(e) Line coverage measured by JaCoCo of the final test suites

(f) Branch coverage measured by JaCoCo of the final test suites

(g) Mutation score measured by PITest of the final test suites

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

Figure 5: Differences between *control*, *notifications*, *treatment* and *maximizing* groups



(a) The number of achievement levels reached

Variable	r^2	p-value
Line Coverage	0.21	< 0.001
Branch Coverage	0.23	< 0.001
Mutation Score	0.22	< 0.001
Number of Tests	0.39	< 0.001

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

Figure 6: The number of achievement levels and their Pearson correlations with different test suite metrics

of 44 %, which is significantly lower ($p < 0.01$). The *notifications* group shows no significant differences ($p = 0.76$) compared to the *control* group, again confirming the influence of gamification; the *maximizing* group even achieves slightly higher mutation scores ($p = 0.01$) than the *treatment* group.

Summary (*RQ 2*): *IntelliGame* has a significant influence on the resulting test suites, increasing the number of tests as well as the code coverage and mutation scores.

4.4 RQ 3: Do achievement levels reflect differences in test suites and activities?

RQ1 demonstrated that participants of the *maximizing* group engage most with testing activities, followed by the *treatment* group, then the *notifications* group, and finally the *control* group, who motivated the least with testing. Figure 6a compares how many achievement levels the participants of the four groups reached, and confirms this ranking of the groups with a mean of 2.6 for the *control* group, 4 for the *notifications* group, 6.1 for the *treatment* group, and 15.4 for the *maximizing* group; Fig. 4f shows that this holds throughout the entire duration of the experiment. All groups reached significantly more levels than the *control* group (*notifications* $p < 0.01$, *treatment* $p < 0.01$, *maximizing* $p < 0.01$). Thus, developers who have been awarded achievements indeed are more committed to testing. To see whether developers with more achievements and higher achievement levels also produce overall better test suites, Fig. 6b shows the Pearson rank correlations between test suite metrics (RQ2) and achievement levels. There is a significant moderate correlation between the number of tests and achievements and a significant weak positive correlation for all other metrics. This is a strong affirmation of the gamification approach, as it shows that acquiring achievements is indeed related to producing better test suites.

Summary (*RQ 3*): Higher achievement levels are indicators of better motivation with testing and better resulting test suites.

4.5 RQ 4: Does *IntelliGame* influence the functionality of resulting code?

Figure 4e shows the number of passed tests of the golden test suite during the experiment, where notably the *treatment* and *maximizing* groups exhibit passing tests earlier compared to the other groups. Particularly, the *maximizing* group reached a plateau where most tests passed at minute 43, indicating that solving achievements leads to faster identification and resolution of program faults as well as faster implementation of new functionality.

At the end of the experiment, the average number of failing tests of the golden test suite (Fig. 5h) is 0.9 for the *treatment* group, and 1.6 for the *control* group, with no statistically significant ($p = 0.45$) difference. The *maximizing* and *notifications* groups also show similar results to the other groups, although participants in the notifications group have the highest average number of failing tests (2.25). The lack of significant differences can be attributed to several factors: (1) the experiment duration was long enough for participants to complete the implementation even with inferior testing behavior, (2) the golden test suite consisted of only six test cases, and (3) the two methods to be implemented allowed for a limited number of distinct possible bugs.

Summary (*RQ 4*): The achievements of *IntelliGame* lead to earlier implementation of functionality, although we observed no significant differences in functionality at the end of the experiment.

4.6 RQ 5: Does *IntelliGame* influence the developer experience?

According to the exit survey (Fig. 7), the target class was well chosen. All groups claim that they had enough time to implement and test the class and that it was easy to understand and implement. Notable and significant differences can be observed on whether participants believe to have tested their code actively: All participants of the *treatment* group claim to have actively tested their code, while only 55 % of the *control* group did this; this difference is significant ($p = 0.03$). Participants of the *treatment* group are also more confident in that they produced a good test suite than participants of the *control* group, although the difference is not significant ($p = 0.16$). Participants of the *maximizing* group, who invested more time into testing, believe even more often to have produced a good test suite, significantly so when compared to the *notifications* ($p = 0.01$) and *control* ($p = 0.025$) groups, but not against the *treatment* group ($p = 0.14$). Throughout the survey, we generally observe favorable responses by the *treatment* group, although not statistically significant for the other cases. Interestingly, members of the *maximizing* group are least confident in that they produced a good implementation, which indicates a disadvantage of their focus on the achievements rather than the programming task.

Considering the experiences of the *treatment* and *maximizing* groups, who both used achievements (Fig. 8), more than half the participants perceived that *IntelliGame* improved their programming behavior, and more than 60 % were driven by notifications, including both encouraging and progress notifications. Most of the participants understood the individual achievements and how to achieve them. Nearly 60 % would like to use *IntelliGame* in their everyday work. More than 75 % of the participants claim to have been motivated to improve testing, which indicates that the plugin served its purpose.

Summary (*RQ 5*): Users of *IntelliGame* perceived the task as easier, are more confident about their results, and confirm that the achievements motivated them to apply more testing.

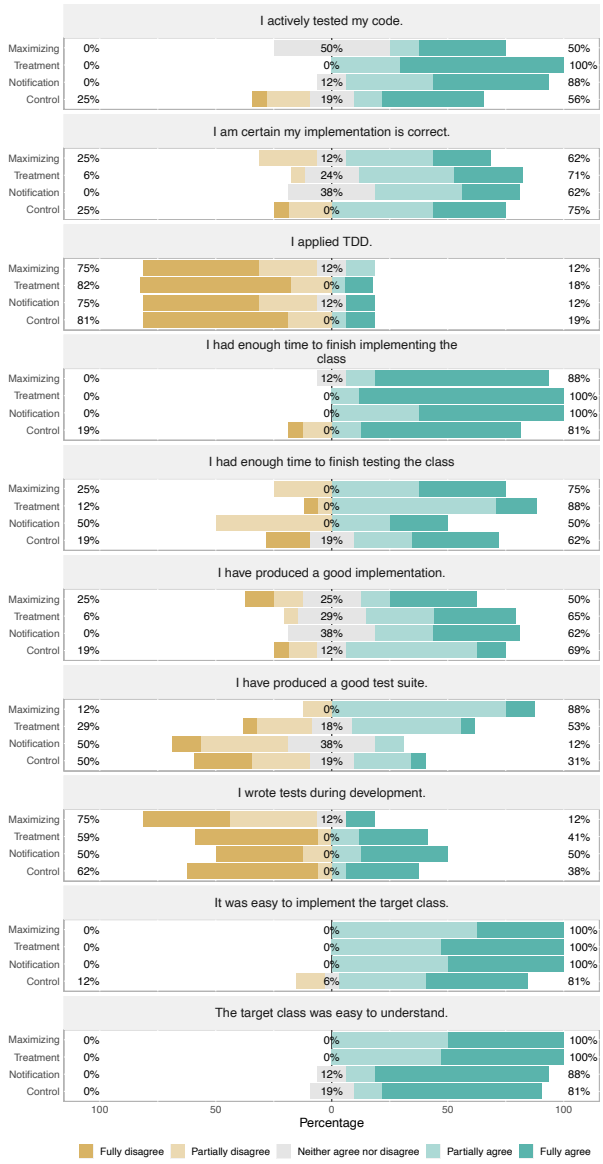


Figure 7: General survey responses

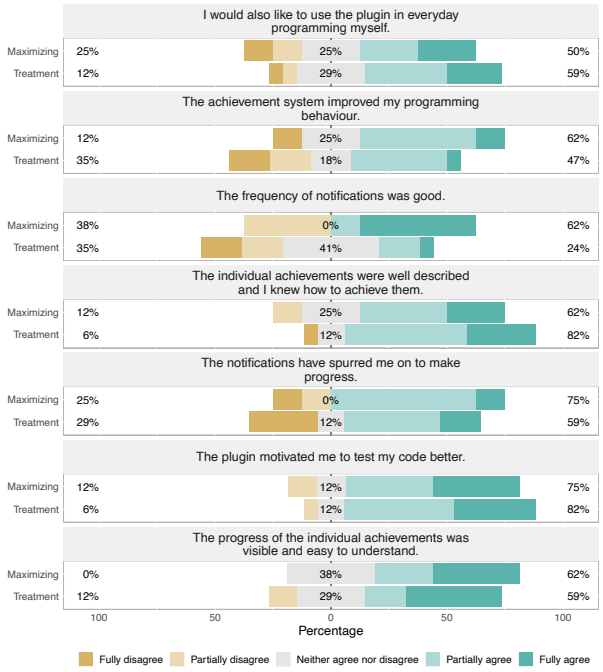


Figure 8: Answers regarding the *IntelliGame* plugin

5 CONCLUSIONS

IntelliGame adds test achievements to the IntelliJ IDE. In an experiment with 49 participants, we found that these achievements significantly improved developers' behavior with respect to writing tests, running and using these tests, and analyzing them. We believe this approach is especially applicable in short-term scenarios like onboarding new staff, fostering a better testing culture in existing teams, or introducing new techniques, since *IntelliGame* only needs to be used until testing practices are understood or adopted. Nevertheless it is similarly conceivable that test achievements become a permanent part of developers' coding environment. However, further research is required to assess the long-term effectiveness of *IntelliGame* and the potential to inspire engagement in developers, as player retention likely becomes a challenge [21].

There are ample opportunities for further improving *IntelliGame* or adding new achievements. For example, our study participants provided suggestions such as rewarding early testing to promote Test-Driven Development, or adapting achievements to individual developers by adjusting achievement levels based on the currently opened project in the IDE or a developer's performance at coding or testing. Furthermore, there is potential to include new game elements and foster competition among developers, such as through a leaderboard—which may be particularly useful when aiming at long-term use of gamification.

In order to support experiment replications and further research on gamifying developer behavior in the IDE, all our experiment material is available at:

https://doi.org/10.6084/m9.figshare.22353352.v1

The source code of *IntelliGame* is available at:

https://github.com/se2p/IntelliGame

6 ACKNOWLEDGMENTS

We would like to thank Jonas Lerchenberger for the initial implementation of *IntelliGame* and his help during the experiments. This work is supported by the DFG under grant FR 2955/2-1, "QuestWare: Gamifying the Quest for Software Tests".

REFERENCES

[1] 2021. JVM ecosystem report 2021. https://snyk.io/jvm-ecosystem-report-2021/. Accessed: 29.03.2023.

[2] David Afshartous and Richard A. Preston. 2010. Confidence intervals for dependent data: Equating non-overlap with statistical significance. *Comput. Stat. Data Anal.* 54, 10 (2010), 2296–2305. https://doi.org/10.1016/j.csda.2010.04.011

[3] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2021. Software professionals' information needs in continuous integration and delivery. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song (Eds.). ACM, 1513–1520. https://doi.org/10.1145/3412841.3442026

[4] Farhan Alebeisat, Zaid Alhalhouli, Tamara Alshabatat, and T.I. Alrawashdeh. 2018. Review of Literature on Software Quality. 8 (10 2018), 32–42.

[5] Andrea Arcuri, José Campos, and Gordon Fraser. 2016. Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 401–408. https://doi.org/10.1109/ICST.2016.44

[6] Mark Ardis, David Budgen, Gregory W. Hislop, Jeff Offutt, Mark Sebern, and Willem Visser. 2015. SE 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. *Computer* 48, 11 (2015). https://doi.org/10.1109/MC.2015.345

[7] Carlos Futino Barreto and César França. 2021. Gamification in Software Engineering: A literature Review. In *14th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 105–108. https://doi.org/10.1109/CHASE52884.2021.00020

[8] 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*. 43–47.

[9] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.* 45, 3 (2019), 261–284. https://doi.org/10.1109/TSE.2017.2776152

[10] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 179–190. https://doi.org/10.1145/2786805.2786843

[11] 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.

[12] 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, SAST 2018, Sao Carlos, Brazil, September 17-21, 2018*. ACM, 39–48. https://doi.org/10.1145/3266003.3266007

[13] 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, SBQS 2019, Fortaleza, Brazil, October 28 - November 1, 2019*, Adriano Bessa Albuquerque and Ana Luiza Bessa de Paula Barros (Eds.). ACM, 178–187. https://doi.org/10.1145/3364641.3364661

[14] 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. *J. Syst. Softw.* 173 (2021), 110870. https://doi.org/10.1016/j.jss.2020.110870

[15] Ronnie Edson de Souza Santos, Cleyton Vanut Cordeiro de Magalhães, Jorge da Silva Correia-Neto, Fabio Queda Bueno da Silva, Luiz Fernando Capretz, and Rodrigo Souza. 2017. Would You Like to Motivate Software Testers? Ask Them How. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*. IEEE Computer Society, 95–104. https://doi.org/10.1109/ESEM.2017.16

[16] Anca Deak, Tor Stålhane, and Guttorm Sindre. 2016. Challenges and strategies for motivating software testing personnel. *Inf. Softw. Technol.* 73 (2016), 1–15. https://doi.org/10.1016/j.infsof.2016.01.002

[17] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart E. Nacke. 2011. From game design elements to gamefulness: defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, MindTrek 2011, Tampere, Finland, September 28-30, 2011*, Artur Lugmayr, Heljä Franssila, Christian Safran, and Imed Hammouda (Eds.). ACM, 9–15. https://doi.org/10.1145/2181037.2181040

[18] V Rama Devi. 2009. Employee engagement is a two-way street. *Human resource management international digest* (2009).

[19] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. 2007. Bug hunt: Making early software testing lessons engaging and affordable. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 688–697.

[20] Fabian Fagerholm et al. 2015. Software developer experience: Case studies in lean-agile and open source environments. (2015).

[21] Shabih Fatima, Juan Carlos Augusto, Ralph Moseley, and Povilas Urbonas. 2021. Gamification for Healthier Lifestyle - User Retention. In *Service-Oriented Computing - ICSOC 2021 Workshops - AIOps, STRAPS, AI-PA and Satellite Events, Dubai, United Arab Emirates, November 22-25, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13236)*, Hakim Hacid, Monther Aldwairi, Mohamed Reda Bouadjenek, Marinella Petrocchi, Noura Faci, Fatma Outay, Amin Beheshti, Lauritz Thamsen, and Hai Dong (Eds.). Springer, 217–227. https://doi.org/10.1007/978-3-031-14135-5_17

[22] Ronald A Fisher. 1956. Statistical methods and scientific inference. (1956).

[23] Alberto César Cavalcanti França. 2014. A theory of motivation and satisfaction of software engineers. (2014).

[24] A. César C. França, Helen Sharp, and Fabio Q. B. da Silva. 2014. Motivated software engineers are engaged and focused, while satisfied ones are happy. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, Maurizio Morisio, Tore Dybå, and Marco Torchiano (Eds.). ACM, 32:1–32:8. https://doi.org/10.1145/2652524.2652545

[25] César França, Fabio Q. B. da Silva, and Helen Sharp. 2020. Motivation and Satisfaction of Software Engineers. *IEEE Transactions on Software Engineering* 46, 2 (2020), 118–140. https://doi.org/10.1109/TSE.2018.2842201

[26] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019. Gamifying a Software Testing Course with Code Defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quiñones, Sarah Heckman, and Jian Zhang (Eds.). ACM, 571–577. https://doi.org/10.1145/3287324.3287471

[27] Yujian Fu and Peter J Clarke. 2016. Gamification-based cyber-enabled learning environment of software testing. *submitted to the 123rd American Society for Engineering Education (ASEE)-Software Engineering Constituent* (2016).

[28] Tommaso Fulcini, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2023. A Review on Tools, Mechanics, Benefits, and Challenges of Gamified Software Testing. *Comput. Surveys* (feb 2023). https://doi.org/10.1145/3582273

[29] Vahid Garousi, Austen Rainer, Per Lauvås Jr., and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *J. Syst. Softw.* 165 (2020), 110570. https://doi.org/10.1016/j.jss.2020.110570

[30] D. Geer. 2005. Eclipse becomes the dominant Java IDE. *Computer* 38, 7 (2005), 16–18. https://doi.org/10.1109/MC.2005.228

[31] Sander Hermsen, Jeana Frost, Reint Jan Renes, and Peter Kerkhof. 2016. Using feedback through digital technology to disrupt and change habitual behavior: A critical review of current literature. *Computers in Human Behavior* 57 (2016), 61–74. https://doi.org/10.1016/j.chb.2015.12.023

[32] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 955–963.

[33] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[34] Edward L. Jones. 2001. Integrating testing into the curriculum - arsenic in small doses. In *Proceedings of the 32rd SIGCSE Technical Symposium on Computer Science Education, SIGCSE*. https://doi.org/10.1145/364447.364617

[35] PK Kapur, AK Shrivastava, and Ompal Singh. 2017. When to release and stop testing of a software. *Journal of the Indian Society for Probability and Statistics* 18, 1 (2017), 19–37.

[36] Lawrence W Kenny, Lung-Fei Lee, GS Maddala, and Robert P Trost. 1979. Returns to college education: An investigation of self-selection bias based on the project talent data. *International Economic Review* (1979), 775–789.

[37] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 61–70. https://doi.org/10.1109/ICSE-SEIP.2019.00015

[38] Herb Krasner. 2022. THE COST OF POOR SOFTWARE QUALITY IN THE US: A 2022 REPORT. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report. Accessed: 29.03.2023.

[39] 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

[40] Mika V. Mäntylä and Kari Smolander. 2016. Gamification of Software Testing - An MLR. In *Product-Focused Software Process Improvement - 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10027)*, Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen-Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen (Eds.). 611–614. https://doi.org/10.1007/978-3-319-49094-6_46

[41] Alok Mishra and Ziadoon Otaiwi. 2020. DevOps and software quality: A systematic mapping. *Computer Science Review* 38 (2020), 100308. https://doi.org/10.1016/j.cosrev.2020.100308

[42] Reza Meimandi Parizi. 2016. On the gamification of human-centric traceability tasks in software testing and coding. In *14th IEEE International Conference on Software Engineering Research, Management and Applications, SERA 2016, Towson, MD, USA, June 8-10, 2016*, Yeong-Tae Song (Ed.). IEEE Computer Society, 193–200. https://doi.org/10.1109/SERA.2016.7516146

[43] K Pearson. 1895. Notes on regression and inheritance in the case of two parents proceedings of the royal society of london, 58, 240-242. *K Pearson* (1895).

[44] Ragunathan Rajkumar, Michael Gagliardi, and Lui Sha. 1995. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *1st IEEE Real-Time Technology and Applications Symposium, Chicago, Illinois, USA, May 15-17, 1995*. IEEE Computer Society, 66–75. https://doi.org/10.1109/RTTAS.1995.516203

[45] Karen Robson, Kirk Plangger, Jan H Kietzmann, Ian McCarthy, and Leyland Pitt. 2015. Is it all a game? Understanding the principles of gamification. *Business horizons* 58, 4 (2015), 411–420.

[46] Richard M Ryan and Edward L Deci. 2000. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American psychologist* 55, 1 (2000), 68.

[47] Simon André Scherr, Frank Elberzhager, and Konstantin Holl. 2018. Acceptance testing of mobile applications: automated emotion tracking for large user groups. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, Christine Julien, Grace A. Lewis, and Itai Segall (Eds.). ACM, 247–251. https://doi.org/10.1145/3197231.3197259

[48] Frank Philip Seth, Ossi Taipale, and Kari Smolander. 2014. Organizational and customer related challenges of software testing: An empirical study in 11 software companies. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–12.

[49] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. 2018. How Do Automatically Generated Unit Tests Influence Software Maintenance?. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 250–261. https://doi.org/10.1109/ICST.2018.00033

[50] Davide Spadini. 2018. Practices and tools for better software testing. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 928–931. https://doi.org/10.1145/3236024.3275424

[51] Andreas Spillner and Tilo Linz. 2019. *Basiswissen Softwaretest: Aus-und Weiterbildung zum Certified Tester–Foundation Level nach ISTQB®-Standard*. dpunkt. verlag.

[52] Klaas-Jan Stol, Mario Schaarschmidt, and Shelly Goldblit. 2022. Gamification in software engineering: the mediating role of developer engagement and job satisfaction. *Empir. Softw. Eng.* 27, 2 (2022), 35. https://doi.org/10.1007/s10664-021-10062-w

[53] Philipp Straubinger and Gordon Fraser. 2022. Gamekins: Gamifying Software Testing in Jenkins. In *44th 2022 IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 85–89. https://doi.org/10.1109/ICSE-Companion55297.2022.9793789

[54] P Suff and P Reilly. 2008. Going the Extra Mile. The relationship between reward and employee engagement. *Institute for Employment Studies, University of Sussex Campus, UK* (2008).

[55] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[56] 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.

[57] Pradeep Kashinath Waychal and Luiz Fernando Capretz. 2016. Why a Testing Career Is Not the First Choice of Engineers. *CoRR* abs/1612.00734 (2016). arXiv:1612.00734 http://arxiv.org/abs/1612.00734

[58] Elaine J. Weyuker, Thomas J. Ostrand, JoAnne Brophy, and Rathna Prasad. 2000. Clearing a Career Path for Software Testers. *IEEE Software* 17, 2 (2000), 76–82. https://doi.org/10.1109/52.841696

[59] Zornitsa Yordanova. 2019. Educational Innovations and Gamification for Fostering Training and Testing in Software Implementation Projects. In *Software Business - 10th International Conference, ICSOB 2019, Jyväskylä, Finland, November 18-20, 2019, Proceedings (Lecture Notes in Business Information Processing, Vol. 370)*, Sami Hyrynsalmi, Mari Suoranta, Anh Nguyen-Duc, Pasi Tyrväinen, and Pekka Abrahamsson (Eds.). Springer, 293–305. https://doi.org/10.1007/978-3-030-33742-1_23

[60] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (1997), 366–427. https://doi.org/10.1145/267580.267590