Gamifying a Software Testing Course with Code Defenders

Gordon Fraser, Alessio Gambi, Marvin Kreis
University of Passau
Passau, Germany
{gordon.fraser,alessio.gambi}@uni-passau.de
kreis03@gw.uni-passau.de

José Miguel Rojas
University of Leicester
Leicester, United Kingdom
j.rojas@leicester.ac.uk

ABSTRACT

Software testing is an essential skill for software developers, but it is challenging to get students engaged in this activity. The Code Defenders game addresses this problem by letting students compete over code under test by either introducing faults (“attacking”) or by writing tests (“defending”) to reveal these faults. In this paper, we describe how we integrated Code Defenders as a semester-long activity of an undergraduate and graduate level university course on software testing. We complemented the regular course sessions with weekly Code Defenders sessions, addressing challenges such as selecting suitable code to test, managing games, and assessing performance. Our experience and our data show that the integration of Code Defenders was well received by students and led them to thoroughly practice testing. Positive learning effects are evident as student performance improved steadily throughout the semester.

CCS CONCEPTS

• Social and professional topics → Software engineering education; **• Software and its engineering → Software testing and debugging**;

KEYWORDS

Software testing education, mutation analysis, testing game, software engineering education, unit testing

ACM Reference Format:
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 '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287471

1 INTRODUCTION

Software testing – the activity of systematically executing a program with the aim of finding faults – is an essential skill for software developers, but it is one they are often not well prepared for. The 2017 Software Fail Watch report [18] describes $1.7 trillion in lost revenue due to software problems, some of which could have been avoided with a more thorough approach to testing. This problem is rooted in education, where testing unfortunately is a relatively neglected component of the computer science curriculum [3], and even when it is taught, then engaging students is challenging. Common testing theory is heavy on management aspects and lighter on actual hands-on testing experience. As a result, students perceive testing as boring and repetitive, and do not acquire the practice and experience that software testing requires.

Gamification is recently heralded as a means to improve testing, both in education [1, 5] and practice [4, 7, 15]. In the wake of this trend, the Code Defenders game [16] aims to engage students with software testing in a competitive and entertaining way. Students compete over program code under test by either introducing software defects that evade the existing test suite (“attacking”), or by improving the test suite to fend off these attacks (“defending”). There is evidence that the game is engaging and players write better tests than outside the game [17], and there is anecdotal evidence of individual Code Defenders sessions being used to spice up university courses. However, to really provide learning effects we argue that Code Defenders should be integrated more tightly as a semester-long activity of software engineering and testing courses.

In this paper, we describe our integration of Code Defenders into a course on software testing taught at undergraduate and graduate level. Such an integration causes numerous challenges: There are *organisational* challenges, such as managing games with fluctuations in student attendance; *technical* challenges, such as scaling up the game to large classes and coping with students trying to find ways to cheat or break the system; and *pedagogical* challenges, such as ensuring fairness and grading student participation. We describe how we overcame these challenges, and use the data collected from the first instance of the course, taken by 123 students who played a total of 197 team-based games, to provide detailed insights into the behaviour of the students, their engagement, and their satisfaction. The data from our experience provides the following insights:

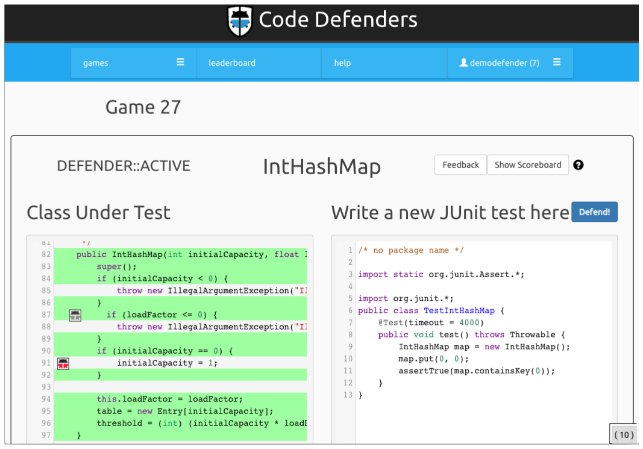
RQ1: How do students engage with the game? Students actively participate and practice their testing skills. On average, students wrote 120.54 tests (with a maximum of 350), and inserted 158.66 artificial faults (maximum of 515).

RQ2: Does student performance improve over time? Students improved their testing performance throughout the course. We observed a moderate correlation of 0.51 between the number of sessions played and the quality of the tests.

RQ3: Does student engagement correlate with exam grades? We observed a moderate correlation of -0.58 between the number of tests written in the game and exam marks, suggesting that better students are more active players.

RQ4: Do students appreciate using Code Defenders in class? In a survey students commented very favourably about their experience, and agreed that the integration is useful and fun.

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287471



```
    */
    public IntHashMap(int initialCapacity, float l
        super();
        if (initialCapacity < 0) {
            throw new IllegalArgumentException("Il
        }
          if (loadFactor <= 0) {
            throw new IllegalArgumentException("Il
        }
        if (initialCapacity == 0) {
            initialCapacity = 1;
        }

        this.loadFactor = loadFactor;
        table = new Entry[initialCapacity];
        threshold = (int) (initialCapacity * loadF
    }
```

```
/* no package name */

import static org.junit.Assert.*;

import org.junit.*;
public class TestIntHashMap {
    @Test(timeout = 4000)
    public void test() throws Throwable {
        IntHashMap map = new IntHashMap();
        map.put(0, 0);
        assertTrue(map.containsKey(0));
    }
}
```

Figure 1: The Code Defenders game in action.

2 THE CODE DEFENDERS GAME

Two challenging activities in software testing are (1) assessing how well a program is tested by the current test suite, and (2) improving the test suite by creating additional tests. Code Defenders [16] is a game intended to engage students in these activities in the context of a Java object-oriented class under test and its test suite: *Attackers* aim to introduce artificial bugs into the class under test that reveal weaknesses in the test suite, while *defenders* aim to improve the test suite by adding tests. Since this process resembles the software testing technique of mutation analysis [2], the artificial bugs are called *mutants*. If a test which passes on the original program fails on a mutant, then that mutant is detected and the defender scores points, whereas the attacker scores points if the mutant is not detected. The number of points a mutant is worth depends on the number of tests it "survives", which further encourages players to create as subtle as possible mutants, and as strong as possible tests.

Attackers can create mutants that are syntactically different but semantically equivalent to the original program. In mutation analysis, this is known as the *equivalent mutant* problem [10], and equivalent mutants cannot be detected by tests by definition. Equivalent mutants may be used by attackers by accident, or intentionally as part of their strategy. If defenders cannot succeed in detecting a mutant and suspect it might be equivalent, then they can flag the mutant and trigger an "equivalence duel". This equivalence duel has to be resolved by the attacker who created the mutant, either by proving non-equivalence by providing a test that detects the mutant, or by revealing their bluff and accepting that the mutant is indeed equivalent – thus losing all points earned for the mutant.

Code Defenders is implemented as a web-based game (shown in Figure 1), and can be played in teams or pairs. Players get to see the source code of the Java class under test, with colour highlighting to indicate the coverage of the defenders' test suite, and with bug-shaped icons labelling the locations and status of the attackers' mutants. Attackers can create mutants by editing the source code of the Java class, and defenders have a code editor to write tests using the JUnit [11], which is the de-facto standard test automation framework for the Java programming language. Besides the information displayed with the class under test (code, coverage, and mutant locations), defenders get to see all tests that have been submitted in the game, while attackers see all mutants. There is also a scoreboard that breaks down the game's current score for each team and player and summarises the status of the mutants.

Whenever humans play games, it is natural that they will try to cheat and bend the rules to their own advantage. Code Defenders uses rules, heuristics, and restrictions to avoid unfair mutants and tests. For example, it does not allow addition of new `if`-conditions into the code, because that would make it easy to create mutants that cannot reasonably be detected by a test without knowledge of the change (e.g., by comparing a variable against an arbitrary numeric value that a tester would hardly guess). Similarly, the restrictions prevent tricks with logical operators, bitshift operators, casts, and include some further specialised restrictions. There are fewer restrictions on defender actions; to ensure that players produce tests like they would in practice (e.g., [13]), the number of JUnit-assertions per test is limited, with a default value of two.

By default, Code Defenders counts unresolved equivalence duels as lost for the attacker after a game ends. To prevent that players exploit this (e.g., by waiting until seconds before the end of the game before marking mutants as equivalent), "grace periods", where the possible game actions are gradually reduced, can be used. If the game duration is too small to include grace periods (as in our case), Code Defenders can also be configured to ignore unresolved equivalence duels, but *force* attackers to handle equivalence duels before being allowed to submit new mutants.

3 CODE DEFENDERS IN THE CLASSROOM

3.1 Course Organisation

The context of our Code Defenders integration is a regular software testing course, organised around the standardised International Software Testing Qualification Board (ISTQB) foundation level tester curriculum [8], plus additional material on more advanced testing techniques. Thus, it consists of regular theoretical lectures, and weekly exercise sessions with exercise sheets featuring examples related to the theory, and introducing the technologies implementing the theoretical concepts, including JUnit. We extended this standard course design with weekly two-academic-hour practical session, throughout the whole semester [6]. These practical sessions featured the use of Code Defenders in the computer lab.

Each practical session revolved around one dedicated Java class under test, on which students competed in teams. In the first session each student took the role of attacker and defender once each; in all successive sessions students participated only in one game, alternating between attacker and defender roles weekly. We bootstrapped each session with a couple of minutes of explanations of the class under test, and also provided some examples of how to write tests to speed up the code understanding phase.

A difficult challenge for the instructor is the selection of appropriate Java classes under test. If a class is too complex, it makes it more challenging for defenders to create good tests, thus allowing the attackers to safely mutate untested code, rather than having to think about how to evade the existing tests and reason about the effectiveness and fault finding potential of the existing tests. On the other hand, if a class is too simple, then the defenders can quickly produce a test suite that covers all testable behaviour, leaving the attackers frustrated and with little room to make changes that are

Table 1: Java classes used in the 12 Code Defenders sessions.

ID	Class	Project	LOC	Methods	Fields
1	Lift	Custom	53	12	5
2	Complex	Math4J	103	24	3
3	Rational	Dittrich Java Intro	114	17	3
4	Option	Apache Commons CLI	260	41	13
5	XmlElement	Inspirento	221	33	6
6	SparseIntArray	Android	161	22	5
7	IntHashMap	Apache Commons Lang	148	14	5
8	ByteVector	Objectweb ASM	154	12	3
9	CharRange	Apache Commons Lang	87	17	5
10	CaseInsensitiveString	Squirrel SQL	177	10	8
11	Document	Apache Lucene	113	15	6
12	Options	Apache Commons CLI	139	16	5

not immediately found. Additionally, the complexity of the chosen classes under test gradually needs to increase over the course of the semester, to keep students appropriately challenged while they are improving their testing skills. While there are some standard code metrics that can support the selection (e.g., lines of code), ultimately judging the difficulty of testing a class is down to experience. There are also technical considerations, for example since Code Defenders offers no support for players to explore dependency classes.

Table 1 summarises the classes we used in the 12 sessions. We created the `Lift` class specifically to introduce Code Defenders. For successive sessions we selected real classes from open-source projects to increase motivation. To avoid that students find the original project online and use its tests, we removed context information, e.g., package name. For the two sessions following the introduction session we used classes implementing mathematical concepts familiar to students (`Rational` and `Complex`), thus allowing interesting gameplay to emerge more quickly. For the remaining sessions we selected pairs of classes that are somewhat similar in nature, so that each student would play as attacker and defender on similarly complex classes. Data structures are particularly well suited since they usually have no dependencies, and students are familiar with the general concept. We also included more specialised classes to increase the level of difficulty and challenge the students. In the last two sessions we used classes that had simple dependencies, but replaced the dependencies with a skeleton interface, such that players were forced to use Java mocking frameworks in their tests.

Although students closely followed their game scores in the overall leaderboard, the score is not suitable as a means for assessment since it depends on many factors beyond the abilities of an individual student (e.g., the abilities and actions of the other players in the game or the use of unfair techniques in the game). We made the Code Defenders sessions count for 10% of the overall course grade, and graded them based on the level of participation. Students were expected to participate in at least 10 out of 12 sessions, and would be awarded one point for each active game participation (capped at 10 points). For each Java class we determined thresholds for the number of tests and mutants that represented "active" participation; to determine the number of suitable tests/mutants we cross-checked all tests and mutants on *all* games for the same class to remove any bias caused by the specific game context. A test is counted if it compiles, passes on the original version of the Java class, and detects at least one mutant; a mutant is counted if it is non-equivalent (i.e., detected by at least one test).

3.2 Game Management

There are several challenges when managing regular sessions with Code Defenders. First, it is desirable that only students physically present in the lab participate in the games. Second, fluctuations in student attendance make it difficult to set up games prior to the actual practical sessions. This was made even more difficult by restrictions in our lab size (40 computers only), which required us to run three smaller successive sessions each week (in which some students worked on their own laptops). We therefore started each session by taking attendance, and then creating a set of games only for the students present in that session. We extended Code Defenders with an administrative interface that supports this process.

Once the students present in a lab session are known, games need to be balanced for a number of factors: First, each student should alternate weekly between attacker and defender roles. Second, there is substantial variation in the skills of students, which may negatively affect games. For example, if a particularly strong defender plays a game where the other players are not so skilled, then that defender would dominate the testing tasks. This might frustrate the other defenders who get fewer chances to reveal mutants, and would make it difficult for attackers to score points. To avoid this issue, we ranked students by their game scores achieved in previous games before creating games, such that students with similar scores (and thus hopefully similar abilities) were teamed up.

We aimed for games with 3 attackers vs. 3 defenders since we observed good game dynamics with these team sizes in the past (for smaller teams it takes longer for mutants and tests to properly interact; for larger teams there tend to be players who cannot, or decide not to, keep up with their team mates). However, we allowed for some variation to accommodate for variations in attendance; in particular, we often made teams of weaker students larger to balance out games. We also used latecomers to equilibrate unbalanced games, e.g., by supporting the losing side with an additional player. Our administrative interface allows monitoring of running games, so that we could react to obvious problems. For example, sometimes defending teams would take long before creating the first test, in which case we located the students in the lab and tried to help. Although we aimed to use the full academic two hours (2 × 45 minutes) for the game, we would usually end games slightly earlier. Once a game is finished, students get to see all tests and mutants, allowing them to reflect, which often led to interesting and active debates among groups of students. Effectively, this means that games lasted on average around 70 minutes.

4 EVALUATION

To evaluate whether Code Defenders can be integrated into a testing course in a way that is beneficial and engaging for students, we investigate the following research questions:

RQ1: How do students engage with the game?
RQ2: Does student performance improve over time?
RQ3: Does student engagement correlate with exam grades?
RQ4: Do students appreciate using Code Defenders in class?

4.1 Experimental Setup

We collected data while running the course at *University of Passau* in the winter semester 2017/2018. We deployed our own installation

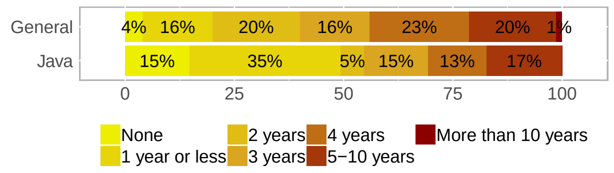


Figure 2: Students' programming experience.

Table 2: Game statistics.

Class	Games	Wins			Averages per Game			
		Att.	Def.	Draw	Players	Tests	Mutants	Duels
1	41	29	12	2	4.02	8.15	25.09	1.26
2	19	15	4	0	6.05	19.53	80.37	5.89
3	15	13	2	0	6.67	42.73	113.00	16.93
4	17	14	2	1	6.47	45.71	84.47	9.24
5	17	10	7	0	6.41	39.41	101.47	6.59
6	14	13	1	0	7.14	45.71	96.50	11.93
7	15	13	2	0	6.47	54.20	86.47	18.13
8	13	12	1	0	6.31	52.62	118.46	12.92
9	13	6	7	0	6.62	64.85	64.23	11.77
10	14	12	2	0	6.21	39.21	100.21	14.57
11	12	9	3	0	6.75	40.83	56.50	10.67
12	7	3	4	0	6.43	30.86	88.14	7.43

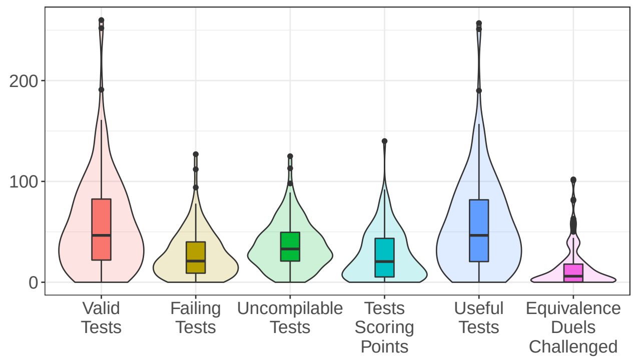


Figure 3: Statistics of defender actions per player; boxplots show quartiles, violinplots show distribution density.

of Code Defenders rather than using the public version, since (1) a large number of students would quickly hit the limits of the public server, (2) we wanted to avoid interactions between students and external remote players, and (3) we wanted to prevent students from creating their own games instead of focussing on the tasks set by us. The course was available to undergraduate and graduate students. In total, it was taken by 123 students (34 female, 89 male), of which 52 were undergraduate students and 71 were at graduate level. The average semester for undergraduate (bachelor) students was 5.77, and for graduate (master) students 1.83, which means that most students were in the final year of their degree. Figure 2 shows the programming experience of the students: 80% of the students declared to have more than two years of general programming experience, while almost half had only recently started using Java (1 year or less) or had not used Java before at all. Consequently, in the initial sessions we had to provide general programming support to some of the students. It is desirable for players to have reasonable Java experience so they can focus on the game.

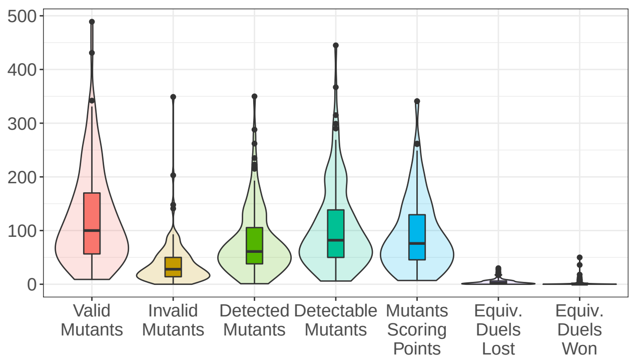


Figure 4: Statistics of attacker actions per player; boxplots show quartiles, violinplots show distribution density.

In total, the students played 197 games on the 12 classes, summarised in Table 2. On the first class, each student played two warm-up games to experience both player roles, resulting in a higher number of games. Because points for active participation were capped at 10, the last class resulted in fewer participants and games. More games were won by attackers; we conjecture that this is because we did not deduct points for unresolved equivalence duels after games ended, as Code Defenders would normally do when using grace periods. On average, games were played with teams of three players each (i.e., 3 vs. 3 games).

4.2 RQ1: Student activities

We assess engagement by analysing the students' actions throughout the course. On average, each student participated in 8.66/12 sessions, and submitted 120.54 tests and 158.66 mutants over all games (average of 19.11 test per game per student, and 35.92 mutants per game per student). These numbers varied across games (see Table 2), with a maximum of 91 tests submitted by a student playing as defender in a single game, and a maximum of 157 mutants submitted by a student playing as attacker in a single game. In total, 18 students did not attend any of the last 6 (out of 12) game sessions, which suggests they abandoned the (optional) course prematurely.

Figure 3 provides more details on the actions performed by students playing as defenders. Although the majority of submitted tests were valid (i.e., compile and pass on the original class under test), a large number of them either had compilation errors or resulted in a failure when run on the original class under test (i.e., had incorrect test oracles). We conjecture that a common strategy to support the understanding of the code is to run a test and observe the behaviour based on the resulting test failure. On average, 37% of all valid tests scored at least one point, i.e., detected a previously undetected mutant. However, on average 98% of the valid tests have the potential to detect a mutant, and are counted as useful during grading. Finally, as expected the number of equivalence duels challenged is much lower than the number of tests submitted per game, although it increased over time as students gained experience.

Figure 4 summarises the actions performed by students when playing as attackers. In contrast to test submissions, the number of erroneous mutant submissions (compilation errors, violations of the constraints on mutations, or duplicate submission of already existing mutants) is much lower than the number of successful

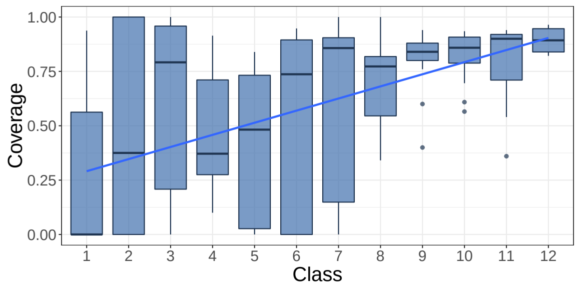


Figure 5: Code coverage achieved throughout the semester.

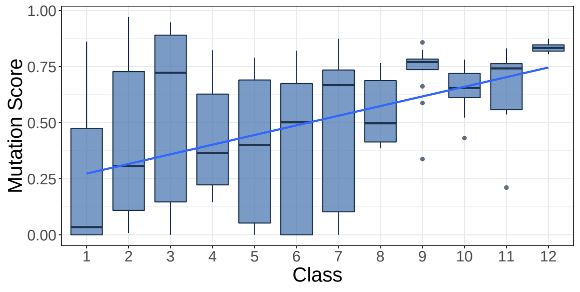


Figure 6: Mutation scores (measured with the Major mutation tool [12]) achieved throughout the semester.

submissions. This is not surprising since a mutant is valid if it follows basic syntactic rules. On average, 64% of the valid mutants were detected by the defenders' tests. By executing all tests from all games on the mutants we can estimate how many could have been detected; on average, 85% of the valid mutants could have been detected, showing that there is ample potential for defenders to further improve; this suggests that defenders might have needed more time, a better way of communication to organising the defense, or that teams of defenders could be made larger. A mutant is only useful if it "survives" at least one test (i.e., is covered but not detected) and thus scores at least one point. On average, 75% of the valid mutants scored at least one point, while the remaining valid mutants either were not covered by any tests, or were accepted as equivalent; the plot suggests that only a small ratio of the mutants were accepted as equivalent. More equivalence duels were lost than won, indicating that attackers either accepted more equivalences or struggled to produce mutant-detecting tests themselves.

Overall our average results indicate that most students engaged well both when playing as attackers and as defenders.

4.3 RQ2: Improvement throughout semester

To determine whether student performance improved throughout the course, we look at the quality of the test suites and mutants resulting from each game. In contrast to RQ1, for this we need an objective measurement of quality that is not dependent on player actions. For tests, the standard metrics are *code coverage* and *mutation score*. Code coverage measures how much of the source code has been executed by a given test suite. Complementary to this, the mutation score measures how good the test suite is at detecting a systematically produced set of mutants. We expect that students playing Code Defenders over time will gain more experience and produce test suites that cover more code and detect more mutants.

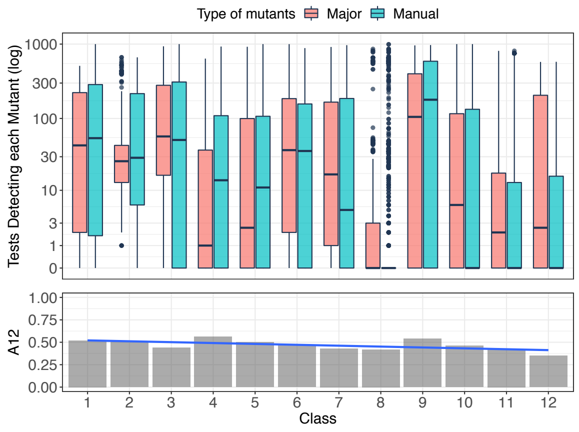
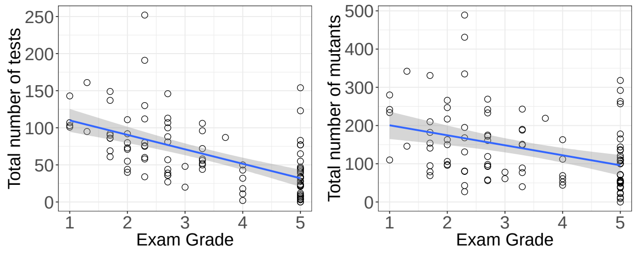


Figure 7: Mutant detection rate throughout the semester.



(a) Test submissions vs. exam grades. (b) Mutant submissions vs. exam grades.

Figure 8: Correlation of exam scores with player actions.

We used JaCoCo [9] to measure branch coverage per game, i.e., on the test suites consisting of all the tests for a game. Figure 5 shows the overall branch coverage achieved per game. Even though the classes under test gradually became more complex and challenging, a clear increase in the coverage throughout the semester is visible, suggesting an improvement in students' testing skills (Pearson's correlation between session number and coverage is 0.51, p-value < 0.001). We measured the mutation scores on the same test suites using the Major mutation tool [12]. Figure 6 confirms the trend already observed in the coverage data (correlation between session number and mutation score is 0.47, p-value < 0.001).

For mutants, determining whether there is an improvement is more challenging, since there is no standard metric of mutant quality. In general, we expect that (detectable) mutants are better if they are more *subtle*, i.e., if there are fewer tests that detect them. However, the tests written as part of the game are influenced by the mutants, which may skew such a measurement. Therefore, we follow the procedure used by Rojas et al. [17] to estimate mutant quality: For each Java class, we use the Randoop [14] test generator to produce 1,000 random tests. Then, we execute each mutant against each random test and count how many tests detect the mutant – the fewer, the harder to detect the mutant is. Since the performance of Randoop may vary across different classes, we use Major [12] to automatically produce a set of baseline mutants on which we also measure the hardness with the 1,000 random tests. This enables us to estimate student advancement in terms of the relative performance of the mutants compared to Major mutants.

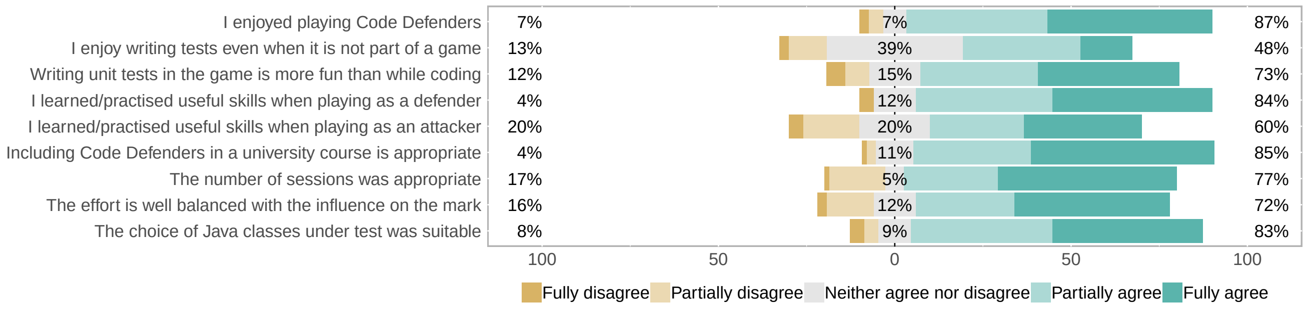


Figure 9: Exit survey responses.

Figure 7 shows the number of tests that detect each of the mutants for the two categories: manual (student-written) and Major mutants). While initially students produced mutants that were detected by more tests, the median mutant quality slowly improved and from week six onwards the student-written mutants are *better* than the ones produced by Major. One exception is the ninth session (class `CharRange`), where the student-written mutants appear to be substantially worse. A closer look reveals that, in this class, changes to certain statements in the constructor lead to state changes that are detected by almost any test, and students happened to mutate these statements quite thoroughly, leading to more easy-to-detect mutants than the Major tool. Figure 7 statistically compares the mutant quality of the two types of mutants using the $\hat{A}_{12}$ Vargha-Delaney effect size measurement; a value of < 0.5 in this case means that the student-written mutants are better. The Pearson correlation between the number of the session and the $\hat{A}_{12}$ effect size is -0.6 (p-value = 0.04), showing that although effect sizes are small, there is a statistically significant improvement throughout the semester.

4.4 RQ3: Game activity vs. exam grades

In order to establish whether there is a relation between player engagement and final exam mark, we calculate the Pearson correlation between test/mutant submissions and the overall grade achieved on the end-of-term written exam. Exam grades are given on a scale of 1–5 (with intermediate steps of .3 and .7), with 1 being the best mark, and 5 being a fail. Figure 8(a) and Figure 8(b) show the correlation between the number of defender actions and mutant submissions, respectively, and the resulting final term exam grade (the exam was taken by 97 students). For both player roles there is a moderate correlation of -0.58 and -0.38, respectively (both p-values < 0.001), confirming that better students are more active players. The correlation is higher for test submissions, which is likely because it is easier to submit a mutant even without fully understanding the code and the tests. While we cannot conclude from this whether the game *influences* the exam marks without further experimentation, the relation of in-game performance and exam performance suggests that including Code Defenders gameplay in the grading is indeed appropriate, which is a prerequisite for making the game a mandatory part of a testing course.

4.5 RQ4: Student satisfaction

After the last Code Defenders session, we ran an anonymous survey among the students, in which we asked them to reflect on their experience and satisfaction with the Code Defenders integration. We announced the survey in-class and via email, and allowed students to voluntarily complete it in their own time. In total, 75 students responded to the survey. Figure 9 summarises the main questions (we omit feedback on how to improve the Code Defenders game itself as it falls out of the scope of this paper), which are overwhelmingly positive: Only two students did not like the integration, and three partially disliked it; the free-text responses suggest that the main points of criticism are limitations of the code editor, performance issues, and a desire to integrate more diverse testing concepts from the lectures into the game sessions, rather than just unit testing.

Although students tend to claim they do like to write tests even outside the game, they confirm it is more fun to do so as part of the game. A vast majority of students felt that they learned useful skills while playing as defender, and while they still agree that they learn useful skills when playing as attacker, there is less agreement compared to the defender role. This is in line with our observations of students submitting mutants with less consideration compared to tests. Students felt that using a game in a university course is appropriate; while the majority of students felt the number of sessions was appropriate, 17% of them disagreed on this. The explanations in the free-text responses reveal that a few students thought the tasks became a bit repetitive, and they would have liked to see more challenging classes over time. Even though the game sessions required no preparation time for the students, 16% of them thought that being awarded 10% of the overall course mark for the game sessions was not appropriately balanced (according to free-text responses it should contribute more), although the majority was satisfied with this marking decision. Finally, on balance the students were satisfied with the choice of classes under test and even suggested more challenging classes could have been used.

5 CONCLUSIONS

Overall, students engaged actively and enjoyed playing Code Defenders as part of our gamified testing course, while at the same time improving their testing skills. This positive outcome encourages us to continue using Code Defenders in the future, and we hope this experience report will inspire and assist other educators to integrate Code Defenders as an educational resource as part of their own software testing courses. Code Defenders is open-source and available on GitHub: https://github.com/CodeDefenders. A public installation is also available to play online at http://code-defenders.org.

ACKNOWLEDGEMENTS

This research has been supported by the Erasmus+ project IMPRESS 2017-1-NL01-KA203-035259 and DFG grant FR 2955/2-1.

REFERENCES

[1] 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*. ACM, 43–47.

[2] Timothy Alan Budd. 1980. *Mutation Analysis of Program Test Data*. Ph.D. Dissertation. Yale University, New Haven, CT, USA.

[3] David Carrington. 1997. Teaching software testing. In *Proceedings of the 2nd Australasian Conference on Computer Science Education*. ACM, ACM, 59–64.

[4] Tommaso Dal Sasso, Andrea Mocci, Michele Lanza, and Ebrisa Mastrodicasa. 2017. How to gamify software engineering. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 261–271.

[5] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. 2007. Bug hunt: Making early software testing lessons engaging and affordable. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. 688–697.

[6] Gordon Fraser, Alessio Gambi, and José Miguel Rojas. 2018. A Preliminary Report on Gamifying a Software Testing Course with the Code Defenders Testing Game. In *Proceedings of the 3rd European Conference of Software Engineering Education*. ACM, 50–54.

[7] Félix García, Oscar Pedreira, Mario Piattini, Ana Cerdeira-Pena, and Miguel Penabad. 2017. A framework for gamification in software engineering. *Journal of Systems and Software* 132 (2017), 21–40.

[8] International Software Testing Qualification Board (ISTQB). 2011. Certified Tester Foundation Level Syllabus. https://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html4.

[9] JaCoCo. 2018. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/. [Online; accessed October 27, 2018].

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

[11] JUnit. 2018. JUnit. https://junit.org/junit4/. [Online; accessed October 27, 2018].

[12] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 433–436.

[13] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[14] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 75–84.

[15] Reza Meimandi Parizi. 2016. On the gamification of human-centric traceability tasks in software testing and coding. In *Software Engineering Research, Management and Applications (SERA), 2016 IEEE 14th International Conference on*. IEEE, 193–200.

[16] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *The 11th International Workshop on Mutation Analysis*. IEEE, 162–167.

[17] José Miguel Rojas, Thomas D White, Benjamin S Clegg, and Gordon Fraser. 2017. Code Defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 677–688.

[18] Tricentis. 2017. Software Fail Watch: 5th Edition. https://www.tricentis.com/software-fail-watch/. [Online; accessed October 27, 2018].