A Preliminary Report on Gamifying a Software Testing Course with the Code Defenders Testing Game

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

Alessio Gambi
University of Passau
Passau, Germany
alessio.gambi@uni-passau.de

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

ABSTRACT

It is challenging to teach software testing in a way that is engaging for students, and to ensure that they practice effective testing sufficiently. Code Defenders is an educational game that is intended to address this problem: Students compete over code under test by either introducing faults (“attacking”) or by writing tests (“defending”). We have integrated Code Defenders as a mandatory component of a software testing course at the University of Passau, which featured ten game sessions of two hours each and involved 120 students. In this paper, we describe how this integration took place and provide some initial insights into our experiences. Code Defenders and the course material are freely available, allowing others to replicate this setup and to gamify their own testing courses.

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, and José Miguel Rojas. 2018. A Preliminary Report on Gamifying a Software Testing Course with the Code Defenders Testing Game. In *ECSEE'18: European Conference of Software Engineering Education 2018, June 14–15, 2018, Seeon/ Bavaria, Germany.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3209087.3209103

1 INTRODUCTION

Software testing is an important aspect of software engineering, but it is not only tedious and error-prone to apply in practice, it is also difficult to teach and a relatively neglected component of the computer science curriculum [2]. The common software testing curriculum, for example described by the International Software Testing Qualifications Board (ISTQB),[1] is heavy on management aspects and lighter on actual hands-on testing experience – due to its intrinsically heuristic nature, learning software testing requires much more practice than is commonly taught. As a result, students are often not optimally prepared to test software effectively, contributing to the dire state of software testing in practice. For example, the 2017 Software Fail Watch report[2] 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.

In order to increase student engagement with software testing, we have recently introduced the Code Defenders online game [3] as part of our ongoing work on gamification of testing [**?**]. In Code Defenders, players compete over a Java class under test by either trying to introduce software defects that evade the existing test suite (“attacking”), or by improving the test suite to fend off these attacks (“defending”). In previous investigations, we have established empirically that the game is engaging and players write better tests than outside the game scenario [4] and we have also explored the use of Code Defenders to teach fundamental software testing concepts [**?**].

In this paper we describe ongoing efforts to integrate Code Defenders into a university course on software testing. Such an integration raises a number of issues that need to be tackled, ranging from the overall course management, classroom management, handling of individual games, and assessment of student performance. We provide preliminary insights into how we solved these issues and some of our findings. We make Code Defenders as well as all course material available freely, in order to help making other software testing courses more exciting, and we hope that it will also see adoption outside university courses.

2 THE CODE DEFENDERS GAME

Code Defenders [4] is an online game, freely available for playing at `http://code-defenders.org`. The current implementation of Code Defenders uses Java as the programming language, and JUnit[3] and Mockito[4] as the framework for writing tests. These are informed design choices: Java is arguably the most popular programming language[5], and JUnit and Mockito are the de facto standard frameworks for unit test automation and mocking in Java.

There are two main game modes: *duel*, where each game consists of one attacker and one defender competing in a turn-based fashion, very much like a traditional board game; and *battle*, where each game consists of a team of defenders and a team of attackers that play without turns. Attackers use a code editor to introduce artificial

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

[2] https://www.tricentis.com/software-fail-watch/

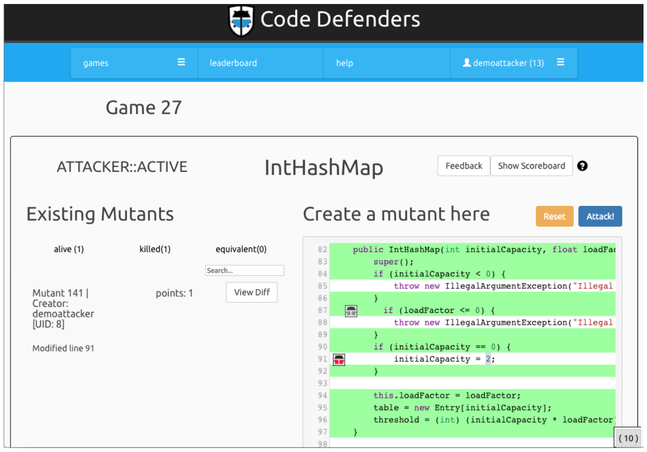
[3] http://junit.org

[4] http://site.mockito.org/

[5] https://www.tiobe.com/tiobe-index/

ECSEE'18, June 14–15, 2018, Seeon/ Bavaria, Germany

ACM ISBN 978-1-4503-6383-9/18/06…$15.00
https://doi.org/10.1145/3209087.3209103



Mutant 141 | Creator: demoattacker [UID: 8]

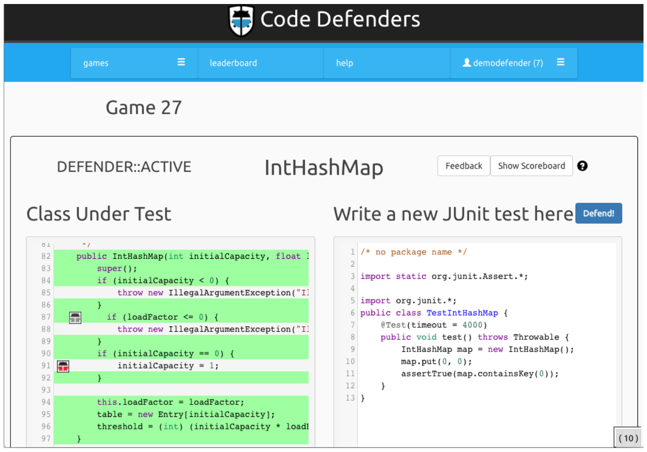
points: 1

Modified line 91

```
82     public IntHashMap(int initialCapacity, float loadFac
83         super();
84         if (initialCapacity < 0) {
85             throw new IllegalArgumentException("Illegal
86         }
87           if (loadFactor <= 0) {
88             throw new IllegalArgumentException("Illegal
89         }
90         if (initialCapacity == 0) {
91             initialCapacity = 2;
92         }
93
94         this.loadFactor = loadFactor;
95         table = new Entry[initialCapacity];
96         threshold = (int) (initialCapacity * loadFactor
97     }
98
```

Figure 1: The attacker view

Attackers have detailed information about all mutants created in the game so far, and use a code editor to create new mutants. The code editor shows the code coverage achieved by the defending team in terms of coloured highlighting.



```
81       */
82     public IntHashMap(int initialCapacity, float l
83         super();
84         if (initialCapacity < 0) {
85             throw new IllegalArgumentException("Il
86         }
87           if (loadFactor <= 0) {
88             throw new IllegalArgumentException("Il
89         }
90         if (initialCapacity == 0) {
91             initialCapacity = 1;
92         }
93
94         this.loadFactor = loadFactor;
95         table = new Entry[initialCapacity];
96         threshold = (int) (initialCapacity * loadI
97     }
```

```
/* 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 2: The defender view.

Defenders have detailed information about all tests created in the game so far, and use a code editor to create new JUnit tests. A code viewer shows the class under test together with coverage highlighting, and bug markers denote where the attackers have introduced mutants.

faults (Figure 1). This resembles the idea of mutation testing [1], where artificial defects are produced automatically. Each "attack" results in a mutant, which is a version of the Java class under test that might contain one or more faults. Defenders use a code editor to write JUnit tests (Figure 2). These tests are then executed on the mutants produced by the attackers, and whenever a test that passes on the class under test but fails on a mutant, that mutant is considered to be detected ("killed") and is ruled out of the game.

A point scoring system is in place to make the game engaging for all players: Attackers receive points whenever a mutant "survives" the execution of a new test; thus, the more tests a mutant survives the higher becomes its score. Defenders receive points whenever a new test detects a previously undetected mutant. The number of points gained by a defender for detecting a mutant equals the number of points accumulated by that mutant. Therefore, killing more resilient mutants results in scoring more points by the defenders. At any time, attackers and defenders can view a scoring table, which breaks down the game's current score for each team and player and summarises the status of the mutants: How many are still alive, how many were killed, and how many are equivalent.

Equivalent mutants represent a particular scenario which can arise if an attacker produces a version of the class under test that is syntactically different, but semantically identical to the original class under test. By definition, equivalent mutants cannot be killed by any test case, and they can be introduced by attackers accidentally, or intentionally as part of their game strategy. If defenders cannot succeed in detecting mutants and suspect those might be equivalent, then they can flag the mutant as suspected equivalent, triggering an "equivalence duel". This puts the onus on the attacker who created that suspected equivalent mutant, who must either prove non-equivalence by providing a test that kills the mutant, or reveal a bluff and accept that the mutant is equivalent. Points are at stake during these equivalence duels.

Two difficulty levels, "easy" and "hard", determine what players of each team get to see in the game. For hard games – hard is the default level we adopted in the course – only partial information is available: Attackers do not get to see the actual tests written by defenders, but only the code coverage they achieve; this coverage is shown using source code highlighting. Defenders do not get to see the actual code changes applied by attackers, but only their location in the source code; this is shown by means of "bug" icons next to changed lines (e.g., see Figure 2). In contrast, in easy games, all this information is disclosed to all players.

3 CODE DEFENDERS IN THE CLASSROOM

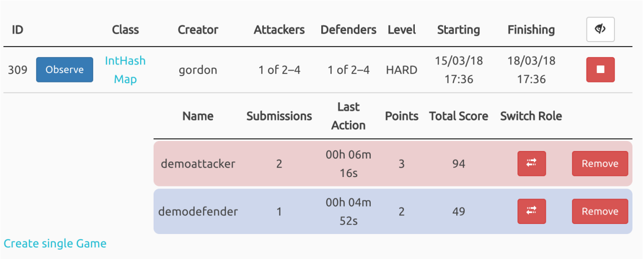
3.1 Context

In the past, we have used Code Defenders for individual motivational lab sessions with students, but we have not previously made Code Defenders an integral part of a university course. In the winter semester 2017/2018 we offered a course on "Software Testing" at the University of Passau, open to both undergraduate and graduate students. The course ran for 15 weeks, and each week consisted of a two-academic-hour lecture, a one-academic-hour exercise session, and a two-academic-hour practical session. The lectures covered the International Software Testing Qualification Board (ISTQB) foundation level tester curriculum [?], plus additional material on more advanced testing techniques. The exercise sessions described examples related to the theory, and introduced the technologies implementing the theoretical concepts, including JUnit and Mockito. The practical sessions featured the use of Code Defenders in the computer lab. The lab available for the course had 40 computers, and to accommodate for the interest in the course we decided to run three groups of 40 students each, for a total of 120 students.

3.2 Technical Setup

The online version of Code Defenders is freely available,[6] as is its source code along with setup instructions.[7] While the public

[6] http://code-defenders.org
[7] https://gitlab.infosun.fim.uni-passau.de/gambi/Code-Defenders-Public



ID		Class	Creator	Attackers	Defenders	Level	Starting	Finishing	
309	Observe	IntHash Map	gordon	1 of 2–4	1 of 2–4	HARD	15/03/18 17:36	18/03/18 17:36	■

Name	Submissions	Last Action	Points	Total Score	Switch Role	
demoattacker	2	00h 06m 16s	3	94	⇄	Remove
demodefender	1	00h 04m 52s	2	49	⇄	Remove

Create single Game

Figure 3: The administrator view.

The administrator view gives an overview of running games, summarising the game state. In this example, there is one attacker (in red) and one defender (in blue). For each player the overview shows the number of points and time of the last action, to help identify struggling students.

installation of Code Defenders is running on a server capable of hosting games for reasonably-sized classes, in the scope of the testing course, we decided to use a local installation for two reasons.

First, given more experience and the focused interactions with Code Defenders over the course of the semester, we expected that students would over time create many more tests and mutants, thus hitting the limits of scalability on the public server: Whenever a new mutant or test is submitted to the game, first it needs to be compiled, and, then, a potentially large number of test executions might follow (each test that covers a mutated line needs to be executed on the mutant.) To avoid this problem, we deployed Code Defenders on the local computing infrastructure at the University of Passau.

Second, the public installation of Code Defenders is free and open to everybody and allows anonymous users to upload new classes, create new games, and create new users. This is undesired in the classroom: We want students to focus on the task set by the instructors without getting distracted playing other games. Similarly, we want to avoid interactions between the students in the lab and other remote players. Therefore, in our locally deployed version of Code Defenders we disabled the features that would allow students to create new users and games and prevented the access to the game from outside the computer lab. To support instructors, we implemented a new administrative interface (Figure 3) to handle all the classroom related aspects described in this paper.

3.3 Classroom Management

Although there was a schedule of students for each session, there are natural fluctuations: Students miss their sessions, arrive late for them, or even request to occasionally join different sessions. Furthermore, we wanted to supervise students during the lab sessions and thus needed to prevent them from joining games online from outside the lab. To solve this problem, we started each lab session by taking attendance and creating a set of games only for the students present in that session.

When creating games, there are several things to take into account: First, there are two player roles (attacker and defender), and students initially tend to prefer starting off as attackers (creating mutants is often perceived as an easier task than writing unit tests). To keep the balance, we decided to have each student alternate the attacking and defending roles weekly. 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 kill mutants, and would make it difficult for attackers to score points. To overcome this problem, we decided to balance teams in terms of the student abilities; that is, we created games where students with similar abilities were teamed up. As an estimate for a player's ability, we used their scores from previous Code Defenders games.

All these considerations mean that the initial setup of the lab session can take several minutes. In particular, the fluctuations of students and latecomers initially severely disrupted the lab sessions. By implementing the administrative interface and improving it over the course of the semester, we drastically reduced the time to setup the session to a few minutes. In theory, stricter handling of attendance fluctuations would also allow to prepare the sessions beforehand, thus removing the setup time entirely.

We aimed for 3 vs. 3 games (i.e., 3 defenders vs. 3 attackers) as anecdotally we observed good game dynamics in the past with these team sizes. However, due to the variations caused by attendance fluctuations and the resulting difficulty to achieve balanced games, we also allowed games with more than three player per side (up to 5 vs. 5), and games with uneven number of attackers and defenders (e.g., 3 vs. 5). Once the infrastructure for handling users had improved enough to dynamically manage games, we utilised latecomers to equilibrate unbalanced games. For example, we supported the losing side with an additional player (although we observed that latecomers tend to be technically weaker students, and thus did not fundamentally affect the gameplay.)

During the game we would monitor game statistics and react to obvious problems. For example, if the defenders did not manage to come up with effective tests we would locate the students in the lab and try to help them write tests. This was made somewhat more complicated because we registered students in Code Defenders using their ids, that is, without keeping track of their real names (except in an external spreadsheet); in the future we will keep track of real names in the game as well. In a few cases we also moved students between running games.[8] For particularly stale games, we sometimes joined as players; that is, we would join as attackers and seed some strong mutants if the attackers did not manage to sufficiently challenge a defender team, or we would join as defenders to seed some example tests if the defenders were struggling or to trigger some equivalence duels to keep attackers busy if they had been flooding the game with many (likely equivalent) mutants.

There were some minor variations in the game time in each lab session, based on the variations in the actual starting time. We tended to end the games ten minutes before the end of the lab session (90 minutes) in order to give students some time to reflect on the games. In particular, once a game is ended, the defenders get to see the actual changes of mutants they had not detected, and that often led to interesting and active debates among groups

[8] In the current implementation of Code Defenders, migrating students across games will reset their score.

of students. Effectively, this means that games lasted on average around 70 minutes.

3.4 Game Management

Whenever humans play games, it is natural that they will try to cheat and bend the rules to their own advantage. It is no surprise that we also observed this during the lab sessions with students. 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 never guess).

In the first few sessions we noticed many new such scenarios we had not anticipated (e.g., tricks with bitshift operators or casting). We observed the reactions of students during the lab, their feedback (defenders often complain about mutants they find unfair, and attackers often show off if they discover a new way to game the system), and the statistics provided by Code Defenders to identify "cheeky" mutants; consequently, we kept improving the restrictions until a fair setting was reached (e.g., confirmed by the defending teams). We also observed some students attempting to use advanced tricks (e.g., reflection) in their tests. However, unlike tricks applied by attackers, these advanced techniques in the tests were mainly explored by particularly strong students who were already good at "normal" testing. As a consequence, we decided that this was not an issue but rather a means to keep strong students engaged.

A further issue was caused by equivalent mutant duels: Normally, any unresolved equivalence duels at the end of the game are considered as accepted equivalences, such that attackers lose the corresponding points of these mutants. Code Defenders uses "grace periods" where the possible game actions are gradually reduced to avoid unfair defensive strategies like claiming equivalence on all remaining mutants seconds before the game ends. For example, in the last couple of minutes it is only possible to resolve equivalence duels, but not to add new mutants or flag more equivalent mutants. However, given the short duration of the lab sessions it seemed infeasible to integrate grace periods. We therefore changed the game behaviour such that attackers would *not* lose the points for unresolved equivalence duels. Further, we changed the game to *force* attackers to handle equivalence duels whenever they arose. That is, attackers were not allowed to submit new mutants until they had resolved all equivalence duels they had received.

3.5 Course Management

A challenging task for the instructor is the selection of suitable classes under test. We started off with a simple artificial example class in the first session to introduce students to Code Defenders; the class under test was simple enough to play two games in the first session, so that students experienced both player roles. For all following sessions, we used classes from open source Java projects, since we find that the use of real code adds to the motivation of students. A possible downside of using open source code is that students might find the original project and use tests written by its developers. We therefore removed context information (e.g., package name, copyright notice, etc.) that would reveal the source of the class.

Currently, Code Defenders does not allow players to explore dependency classes and libraries, and so we restricted the selection to independent classes; i.e., classes that only depended on classes from the Java standard library (including collections, etc.) To grow the pool of candidate classes, we eventually also resorted to refactoring classes to fit with this restriction, e.g., by removing parts of the code that cause dependencies.

At the same time, it is also challenging to find the right level of difficulty: On the one hand, if a class is too large, it would take long until the defenders manage to cover all the code; additionally, the attackers' actions become largely detached from what the defenders do as they always have untested code to "safely" mutate. However, from an educational viewpoint it is preferable if the attackers have to think harder about how to evade the existing tests; this is where we see the main learning effect for attackers, as they reason about the effectiveness and fault finding potential of the existing tests. On the other hand, if a class is too small or too simple, then the defenders would quickly exhaust the testable behaviour, leaving the attackers frustrated and with little room to make changes that are not immediately found. Similarly, the complexity of the class under test needs to increase over the course of the semester, with the testing skills of the students presumably improving over time.

Following the introduction session, we had two sessions using similar mathematical classes (a rational number class and a complex number class). The reason behind this choice was that students are already well familiar with the underlying concepts, and so code comprehension becomes much easier, reducing the time it takes until interesting gameplay emerges. We then selected pairs of classes that are somewhat similar in complexity and nature, so that each student would play as attacker and defender on a similar class in successive weeks. In particular, we found that data structures are well suited for Code Defenders sessions, since data structures tend to have no dependencies, and students are familiar with the general concept of data structures. Nevertheless, we also included more specialised classes (e.g., components of the Apache Lucene[9] search engine) to increase the level of difficulty and challenge the students. Towards the end of the semester we also included classes that had dependencies, but omitted the dependencies (except for a skeleton interface defined as an inner class of the class under test). The reason for this was to force the use of mocking in the tests.

To reduce the time spent on code comprehension, we bootstrapped each session with a couple of minutes of explanations of the class under test for that session, and also provided some examples of how to write tests for the class under test.

3.6 Assessment

The Code Defenders lab sessions were included in the overall course grading. Considering that there is almost no required preparation time for students and the effort mainly consists of attending the weekly sessions, we decided to make the lab sessions count for 10% of the overall grade (which is proportional to the ECTS effort compared to the exam). We ignored the initial three sessions for

[9] https://lucene.apache.org/

grading, and for all remaining sessions determined whether students participated actively. We did not consider the game score for evaluating students, since this 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). To alleviate this problem, we analysed the tests and mutants for each student, and created a full "kill matrix", i.e., record of which test detects which mutant. This is currently not supported in Code Defenders directly, but we plan to integrate the computation of such a kill matrix in a future version of the game.

Since this was an experimental run of the course, we decided not to be too restrictive about when to count participation as active: Attackers were considered active in a session if they produced at least 5 mutants that were not immediately detected, while defenders were considered active in a session if they wrote at least 5 compiling tests that managed to kill some mutants.

4 PRELIMINARY OBSERVATIONS

Over the course of the semester, we have collected data on student performance, engagement and perception. While a detailed analysis of the data remains to be done, we can already provide some initial lessons learned and challenges from our experience.

4.1 Peer Engagement and Communication

It seems that the game succeeded in engaging students at all levels, and we believe that our way to group students by abilities contributed to this end. Anecdotally, strong students managed to challenge each other particularly well, with debates often continuing after the lab sessions. Students struggling with more basic aspects of writing unit tests also benefited from the team-based nature of the game; for example, as defenders they can see the tests written by their team mates and learn from them. Consequently, in the future we plan to improve and extend the communication options available in the game. Although Code Defenders currently features a basic instant messaging service, this feature suffered from technical issues and was not utilised much by the students.

4.2 Engagement Through Competition

During the game play, students see how they perform in terms of their score, and there is a leaderboard in which they see their overall performance compared to their peers. Although leaderboards are sometimes taunted for being an overused gamification mechanism, we found that students paid more attention to the score than we had anticipated. Even though we informed them that the score would not be taken into consideration for grading, we observed some students explicitly trying to boost their scores by developing dubious game strategies which we will investigate further.

4.3 Learning Effects

Over the course of the semester, each student wrote dozens to hundreds of tests and found many injected bugs in realistic code. This outcome is unlikely to happen in a regular testing course or software engineering capstone project. For example, when given traditional tasks such as achieving code coverage, this is typically done only for individual pieces of code, as it is difficult to keep students (and even professional developers) engaged for long in this activity.

While the defender role clearly leads to testing activities, this is less obvious for the attacker role: Ideally, it should force students to think about fault potential and the quality and coverage of existing tests. However, we occasionally observed students producing seemingly random mutations without spending much thought on them. This seems to particularly be the case for students struggling with program comprehension and effective testing. In the future, we will consider adapting the game play, for example by requiring attackers to submit mutants together with a killing test.

4.4 Gradually Increasing Complexity

The repeated use of the game allows to gradually increase complexity of the code under test, and to grow the students' abilities over time. We found that the inclusion of mocking techniques into the game in the last two sessions refreshed interest, and therefore we will investigate integration of other more advanced testing techniques (e.g., integration tests, tests using databases and other external dependencies, etc.) in Code Defenders.

In general, it is difficult to find the right code complexity for both player roles, but it is equally difficult to find increasingly more challenging classes over the course of the semester to keep students engaged. Since the completion of the course, we have added a feedback mechanism to Code Defenders. This way, we can learn what students thought about classes under test and build up a repository of suitable Java classes for re-use in other courses.

5 CONCLUSIONS

On the whole, the students enjoyed the experience and were well engaged, so the overall conclusion is very positive. For this reason, we will use Code Defenders again in future software testing courses.

We invite everyone to try out Code Defenders: It is available freely online on `http://code-defenders.org`, and setting up a local installation for customised use is straightforward.

ACKNOWLEDGEMENTS

This research has been partially financed by the European Commission through Erasmus+ project IMPRESS 2017-1-NL01-KA203-035259.

REFERENCES

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

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

[3] José Miguel Rojas and Gordon Fraser. 2015. Code Defenders: A Mutation Testing Game. In *The 11th International Workshop on Mutation Analysis.* IEEE. To appear.

[4] 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.