eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/116454/

Version: Accepted Version

Proceedings Paper:
Rojas, J.M. and Fraser, G. (2016) CODE DEFENDERS: A Mutation Testing Game. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation (ICSTW). ICST 2016 : IEEE International Conference on Software Testing, Verification and Validation (ICST) 2016, 10/04/2016-15/04/2016, Chicago, Illinois. Institute of Electrical and Electronics Engineers, pp. 162-167. ISBN: 978-1-5090-1826-0.

https://doi.org/10.1109/ICSTW.2016.43

© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works. Reproduced in accordance with the publisher's self-archiving policy.

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

CODE DEFENDERS: A Mutation Testing Game

José Miguel Rojas, Gordon Fraser
Department of Computer Science, The University of Sheffield, Sheffield, United Kingdom
{j.rojas,gordon.fraser}@sheffield.ac.uk

***Abstract*—Mutation testing is endorsed by software testing researchers for its unique capability of providing pragmatic estimates of a test suite's fault detection capability, and for guiding testers in improving their test suites. In practice, however, widespread adoption of mutation testing is hampered because any non-trivial program results in huge numbers of mutants, many of which are either trivial or equivalent, and thus useless. Trivial mutants reduce the motivation of developers in trusting and using the technique, while equivalent mutants are frustratingly difficult to handle. These problems are exacerbated by insufficient education on testing, which often means that mutation testing is not well understood in practice. These are examples of the types of problems that *gamification* aims to overcome by making such tedious activities competitive and entertaining. In this paper, we introduce the first steps towards building CODE DEFENDERS, a mutation testing game where players take the role of an *attacker*, who aims to create the most subtle non-equivalent mutants, or a *defender*, who aims to create strong tests to kill these mutants. The benefits of such an approach are manifold: The game can serve an educational role by engaging learners in mutation testing activities in a fun way. Experienced players will produce strong test suites, capable of detecting even the most subtle bugs that other players can conceive. Equivalent mutants are handled by making them a special part of the gameplay, where points are at stake in duels between attackers and defenders.**

I. INTRODUCTION

Testing is an essential activity in any software development process, with the aim to ensure that software is of a sufficient quality for its intended application. The quality of test suites is usually estimated with unreliable proxy measurements such as line coverage. In contrast, mutation analysis has the unique advantage of measuring not only how much of a program is executed, but it also provides an estimate of how well the test suite performs at detecting faults in the code that is covered. This is achieved by seeding artificial faults, the so called *mutants*, and determining how many of them a test suite can distinguish from the original program, which is typically captured quantitatively in the *mutation score*. Mutants that are not detected can serve to guide a tester in improving the test suite and to improve its fault detection ability.

Although empirical results have confirmed that test suites that are good at detecting mutants are also good at detecting faults [1], [2], mutation testing is still rarely applied in practice. There are multiple different conjectures why this is the case: First, the number of mutants generated for any non-trivial program can be inhibitively large, creating scalability problems. Second, only a few of these mutants are typically useful—many mutants are trivially easy to detect and/or redundant, skewing the mutation score and providing false confidence in test suites. Third, some of the mutants are equivalent, which means that there exists no test case that could distinguish them from the original program. Although testers can in principle write new tests to kill mutants not yet detected, the existence of these types of mutants makes this a frustrating activity. Finally, the concept of mutation testing can be somewhat confusing to developers first introduced to it; this may be influenced by mutation testing not being a well established component of programming and testing education, which in turn may be due to the lack of supporting educational mutation testing tools [3].

Gamification [4] is an approach where difficult tasks are converted to components of entertaining gameplay; the competitive nature of humans is exploited to motivate them to compete and excel at these activities and to apply their creativity. This is known to be beneficial in an educational setting [5], but can also be applied to overcome hard computational problems. Gamification has been successfully applied in domains such as character recognition [6] and language translation [7], and has also been investigated in the context of software engineering, for example to support version control [8] and testing [9] activities. The problems of mutation testing seem well suited for gamification: Generating subtle, non-trivial mutants is a creative task, as is generating efficient test data to detect these mutants. Also, as the equivalent mutant problem is undecidable in general [10] human effort is typically required to solve it.

In this paper, we introduce CODE DEFENDERS, a web-based game that implements the idea of gamification of mutation testing. Two players compete over one program under test: an *attacker*, who tries to create subtle mutants, and a *defender*, who tries to create the best possible test suite. The attacker scores points by creating mutants that are not detected by the test suite; the defender scores points by adding mutant-killing tests. At face value, the player with the higher score in the end is the winner. However, both players hone their testing skills, and the real winner is the developer of the program under test, who gains great tests and subtle mutants. The gamification can also tackle the equivalent mutant problem: If defenders suspect mutants to be equivalent, they can challenge the attacker to a duel, and whoever can prove non-equivalence with a new test or convince the opponent of the equivalence gains extra points.

CODE DEFENDERS is currently an early prototype that we are using to explore how to make mutation testing fun, and to gather feedback from practitioners and other mutation testing researchers. The game targets Java classes and JUnit tests, and provides basic gameplay. However, there remain several challenges to be tackled, ranging from a fair but competitive scoring system, over modes of entertaining gameplay, to scaling the approach up to larger programs.

II. Background

A. Mutation Testing

Mutation testing is a structural software testing technique used to evaluate the fault-detection capabilities of a test suite. In a nutshell, the process consists of seeding artificial faults ("mutants") in the program, and measuring how many of them are found ("killed") by the test suite. Mutants that remain "alive" after the test suite execution can be used as testing targets to create additional tests that kill them, hence enhancing the existing test suite.

Although evidence suggests that mutation testing is effective at finding real faults (e.g., [1], [2]), it is not currently widely adopted by software engineers, due in part to the following two fundamental technical aspects:

(a) *Large number of mutants*. The mutation testing process is driven by a range of *mutation operators*. The broad diversity of mutation operator groups soon leads to producing a high number of mutants even for simple pieces of code, many of which can be trivially killed or are simply redundant. Whereas automated approaches exist to reduce the number of mutants and produce only the most relevant ones (e.g., [11]), this process is computationally expensive and in practice it requires commitment from developers who needs to decide which mutants their test suites should be run against, and which mutants to use as target to produce new tests.

(b) *Equivalent mutants*. Equivalent mutants are arguably one of the main drawbacks of mutation testing: determining if a mutant is equivalent by hand can be a challenging task even for seasoned programmers [12], while doing so automatically is an undecidable problem in general [10], [13]. An equivalent mutant, although syntactically different, is semantically identical to the original program and therefore no test suite is capable of distinguishing between them. In practice, the later equivalent mutants are identified, the more detrimental they become to the overall cost-effectiveness of the technique: they skew the fault-detection effectiveness estimates and unavailing effort is put in trying to create tests to kill them. While several techniques and systems have been developed to reduce or detect equivalent mutants (e.g., [14]–[16]), their success is generally limited to certain types of mutants and human intervention is still required to discern hard-to-kill (or "stubborn") mutants from equivalent ones [17].

Tackling these two intrinsic limitations of mutation testing, at least given the current state of the art in automation, inevitably requires human intelligence, creativity, and experience in mutation testing. Although mutation testing is increasingly finding its way into programming and testing education [3], not least because of the active engagement of members of the mutation testing community, it remains a peripheral, often cursory, part of the testing curriculum. Consequently, its effect on the learning processes of novice programmers and by extension on the performance of professional software developers remains unclear. Are software developers sufficiently trained to apply mutation testing in their day-to-day tasks? Can they choose the most adequate mutation operators depending on the context of the software under test? Can they distinguish equivalent mutants? In the absence of any conclusive empirical study answering these questions, we venture to propose a gamification approach to familiarise developers with the main concepts of mutation testing and to contribute to increasing the adoption level of mutation testing amongst practitioners.

B. Gamification

Gamification is a methodology in which game design elements (competitions with other players, game rules, point scoring, fantasy scenarios, etc.) are applied in non-game contexts in order to make unpleasant or dull tasks more entertaining and rewarding [4]. It can serve as source of formative experiences for educational purposes or to solve problems which are hard to compute but relatively easy for humans to solve [6], [7]. There are several successful examples of gamification for software engineering, where the methodology has been applied mostly to increase the motivation and performance of people participating in software engineering activities [18]. For instance, the popular FindBugs tool (http://findbugs.sourceforge.net/) was gamified in order to motivate developers to remove warnings from their codebase [19] the formal verification of programs was gamified in such a way that verifying programs does not require highly trained professionals and becomes a more cost-effective activity [20] and finally, the web-based game CodeHunt was developed to teach coding at different skill levels [21]. In this paper we argue that mutation testing is also amenable to gamification, and could potentially be an effective way of bridging the gap between mutation testing research and software engineering practice [4], [22].

III. Code Defenders

In this section, we describe a first approach of applying gamification to mutation testing. This approach has been realised as an online game available at http://code-defenders.dcs.shef.ac.uk.

A. Gameplay

Code Defenders is a turn-based mutation testing game for Java classes and JUnit tests. Two players are involved in each game: an attacker and a defender. They compete against each other by, respectively, attacking and defending a Java class under test (CUT) and its test suite. The attacker's role consists in creating variants of a program under test, i.e., *mutants*. Metaphorically, a mutant represents a fault in the CUT, hence an *attack* to the fault-detection capability of the associated test suite. On the other hand, the defender's role consists in creating unit tests that can detect, i.e., *kill*, those mutants. In doing so, the defender strengthens the test suite and protects the class under test from the faults represented by those mutants.

Attackers have the first turn in the game. Once the attacker succeeds in producing a mutant for the class under test, the turn is passed to the defender, who has the chance to defend

against the attacker's mutant. The game develops with rounds of attack and defence.

Equivalent mutants play an important role in the gameplay. When a defender suspects a mutant is equivalent and claims so, the attacker, who created the mutant in the first place, is challenged either to accept the mutant as equivalent, or to counter by providing a test that kills the mutant.

Two levels of difficulty are currently available, *easy* and *hard*. The difference between the two is that in the *easy* mode, the code of all mutants is revealed to the defender in the form of contextualised *diff* reports, whereas in the *hard* mode, only a brief description of the mutant is presented to the defender (e.g., *There was a change in line 18.*). Intuitively, the easy mode allows for a more reactive testing strategy, where defenders would write tests targeted directly at killing mutants according to how they differ from the original CUT. On the other hand, the *hard* mode fosters a more proactive testing strategy, where defenders would try to defend against all possible attacks matching the mutant description.

The CODE DEFENDERS point scoring system was designed to reflect how well the attacker and defender perform at creating strong mutants and effective tests. Points are awarded to both players at the end of each round. Table I summarises a game with player Alice playing as the attacker, and player Bob taking the role of the defender. The example demonstrates all the components of the scoring system:

TABLE I
GAME SCORING SYSTEM EXAMPLE

Round	Alice (attacker)		Bob (defender)		Explanation
	Action	Points	Action	Points	
start	–	0	–	0	Game starts
1	m_1				Alice submits m_1
			t_1		Bob submits t_1
		0		1	t_1 kills m_1
2	m_2				Alice submits m_2
			t_2		Bob submits t_2
		2		1	m_2 survives t_1 and t_2
3	m_3		t_3		Alice submits m_3
					Bob submits t_3
		2		3	t_3 kills m_2 and m_3
4	m_4				Alice submits m_4
			$eq(m_4)$		Bob claims m_4 equivalent
	t_4				Alice submits t_4
		3		3	t_4 kills m_4
5	m_5				Alice submits m_5
			$eq(m_5)$		Bob claims m_5 equivalent
	ok				Alice accepts
		3		4	m_5 marked equivalent
end		3		4	Game ends; Bob wins

1) Upon submission of a new test, the defender gets one point per mutant killed by the new test (rounds 1 and 3).
2) Upon submission of a new mutant, if it survives, the attacker gets one point for each existing test the mutant has survived (round 2).
3) When the defender claims a mutant is equivalent,...
   a) If the attacker is able to submit a killing test, the attacker gets one point (round 4).
   b) If the attacker accepts the mutant is equivalent (round 5), points awarded depend on the difficulty level of the game. In the *easy* level, where the defender can see the actual mutant, the attacker keeps the points collected so far for that mutant and the defender gets one point. In the *hard* level, in contrast, the attacker loses all points accumulated for that mutant and the defender gets two points.

In short, attackers win if they produce mutants that survive defenders' unit tests longer. Alternatively, defenders win if they produce unit tests that kill most mutants soon after they are submitted. The mechanism to assign points in the case of accepted equivalent mutants is intended to make both players reflect about the equivalent mutant problem before submitting a mutant or claiming its equivalence, simply because a reckless action will have an evident impact on the final score.

B. Game Start

Creating a new game is the first available feature for a player of CODE DEFENDERS. To do so, the player must specify a class under test (CUT) by choosing it from a pre-defined list of examples or by uploading a different class. The player must also pick a role (attacker or defender) and set the number of rounds the game is going to last and its level of difficulty.

Once a game is created, it is added to a pool of open games, which are then available to any other user to join. The game is considered active once a second player has joined, and the attacker is given the first turn to play. A round is complete when each player in the game has taken a turn.

C. The Attacker View

In the attacker's view, shown in Figure 1, a text area allows the attacker to make changes to a copy of the original CUT. The attacker's view also shows all the mutants, alive and killed, and tests in the game. Currently, the attacker is free to apply arbitrary changes to the CUT; in the future, we plan to introduce the notion of fault models to assist in the creation of more meaningful mutants. When the attacker submits a new mutant using the "Attack!" button, the system first tries to compile it. If the compilation fails, the attacker is prompted to go back and edit the mutant. Alternatively, if the mutant compiles, all existing tests in the game are executed against it. We distinguish three possible outcomes of executing of a test against the original CUT and a mutant:

1) The test passes on the original CUT and passes on the mutant, which indicates that the mutant has survived.
2) The test passes on the original CUT but fails on the mutant, hence the mutant has been detected and killed.
3) The test fails on the original CUT, in which case it is labeled invalid and ignored in the rest of the game.

For the sake of fairness, some restrictions apply on what constitutes a valid mutant. First, the public interface of the CUT must not be altered. Second, to avoid the creation of

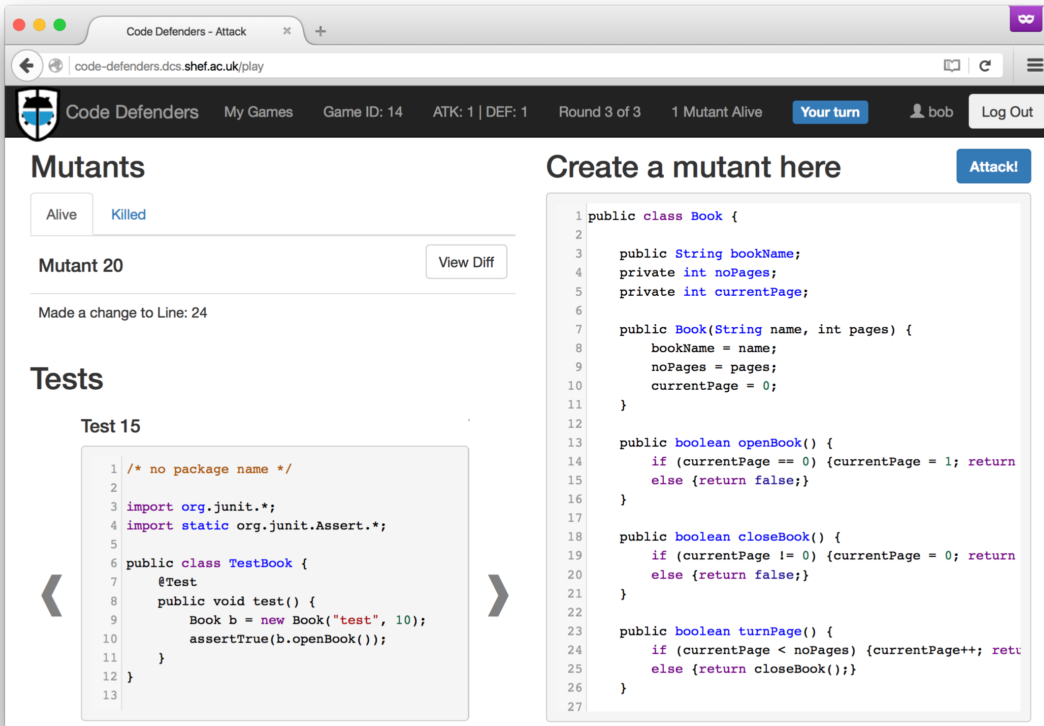

Fig. 1. The Attacker View

mutants that add irrelevant and difficult to find behaviour (e.g., `if(input ==`⟨*some random value*⟩)), new branching or looping statements are not allowed.

D. The Defender View

The defender's role in the game is to write unit tests for the CUT. To facilitate this task, the defender view (Figure 2), consists of a text area with a test template where the player can write a new unit test, a side panel with the original CUT source code, two extra panels with the list of mutants (alive and killed), and the list of tests previously submitted in the game. When the defender clicks on the "Defend!" button to submit a new unit test, a compilation and validation check is run on the test. Only one unit test per submission is allowed (controlled by counting the `@Test` annotations in the test code), and no branching/looping statements are allowed either. If the test is not valid, the defender keeps the turn and can edit the test. Otherwise, the valid test is added to the game and executed against the original CUT and all mutants alive; this action passes the turn to the attacker and finishes a game round.

E. Equivalent Mutant Duels

The system for incorporating equivalency detection takes place over multiple in-game rounds, and assumes some level

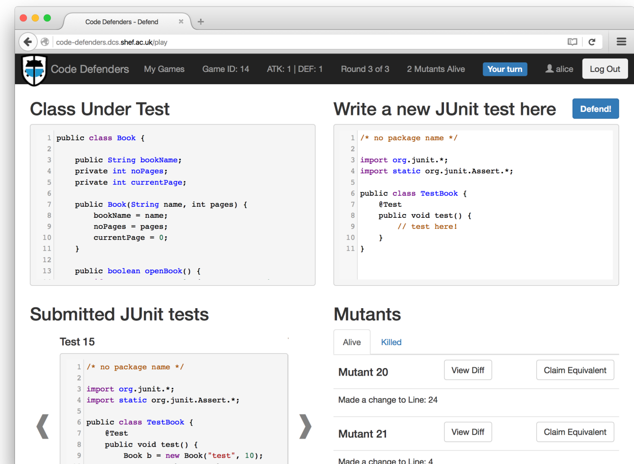

Fig. 2. The Defender View[1]

of competency at creating tests from the attackers, even though their primary task is to create mutants. Defenders can start an equivalence duel when they suspect that a particular mutant may be equivalent and cannot be detected by any test. Attackers must then respond to the challenge in their next turn (Figure 3) by either accepting that the mutant is equivalent (possibly

[1]Due to space constraints, Figures 2-3 simply intend to depict the layout of the different game views; the code they contain is not meant to be readable. Full size screenshots can be browsed at http://code-defenders.dcs.shef.ac.uk.

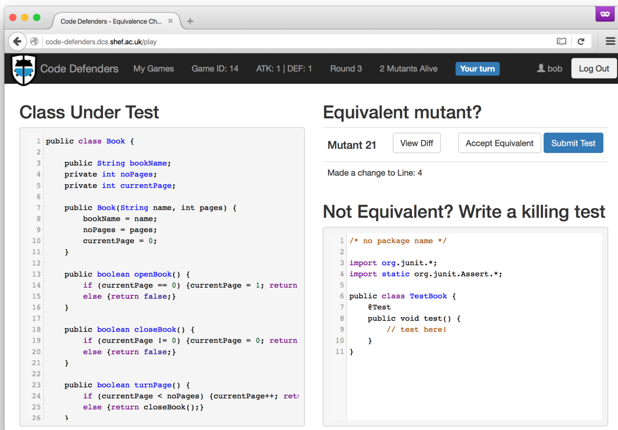

Fig. 3. Equivalent Mutant Challenge View

losing several points) or by submitting a test that would detect it (possibly scoring extra points). This mechanism is expected to make defenders refrain from starting duels unless they truly believe mutants are equivalent.

IV. Open Challenges

Code Defenders is an early prototype that we have developed to explore ways to make mutation testing fun to learn and practice. Several challenges remain to be addressed to provide users with a full entertaining and educational experience, and to explore how this gamification approach can help to overcome the technical problems of mutation testing.

A. Single- and Multi-player Modes

In its current version, Code Defenders does not rely on any test generation or mutation testing tool. Therefore it requires two human players in each game, which limits its playability. In future versions we plan to evolve the game to also support single- and multi-player modes.

A single-player mode implies interaction with an automated opponent. We plan to leverage existing tools to simulate the attacker and defender roles. A mutation testing tool, e.g., Major [23] or MuJava [24], could be used to simulate the attacker role of producing mutants. In the simplest case, the automated attacker could randomly choose and submit pre-computed mutants for the CUT. A more elaborate alternative could involve measuring coverage of the set of tests in the game and attacking with not yet covered mutants. Likewise, existing unit test generation tools for Java, e.g., EvoSuite [25], could be plugged in to simulate the defender role. A soft automated defender could simply pick from a pool of automatically generated tests at random. A harder-to-defeat defender, in contrast, could first run the automatically generated tests on the original CUT and the mutant, which would enable it to submit, when available, killing tests.

A multi-player mode will require Code Defenders to handle multiple players submitting mutants and tests for the same CUTs. Given the current architecture of the game, this will mainly pose engineering challenges on the graphical user interface and underlying database, and a re-design of the game's scoring system.

B. Scoring System

As mentioned earlier, the scoring system of Code Defenders is intended to be fair for the two players involved. That alone, however, does not guarantee an enjoyable playing experience. We expect that our experience with the current version of Code Defenders will help us design a more engaging gameplay and scoring system. One aspect of the scoring system where we see room for experimentation, for instance, is in the handling of equivalent mutants. As explained in the previous section, when a mutant is claimed as equivalent by the defender, the attacker must respond by either accepting the mutant as equivalent, or by providing a killing test. An arguably more fun protocol here could be that when claiming a mutant equivalent, depending on their confidence levels, the defender might be willing to put some points at stake, giving the attacker the options to engage in the challenge, to reject it, or even to bluff about actually having a killing test by topping the amount of points at stake. Eliciting feedback from the software testing community and early adopters of the game will be important to refine the current scoring system, as well as other elements of the game.

C. Scaling to More Complex Classes Under Test

The current interface of Code Defenders only supports the inspection of a single Java class per game. Besides changes to the user interface to support the integration of dependency classes, there are engineering challenges such as decoupling the core game components from the building and testing infrastructure (currently based on Apache Ant and run on the same web server). Furthermore, it is conceivable that other testing techniques could be integrated to support players when using larger code, for example by allowing defenders to measure code coverage on a new test before submitting it.

D. Abstracting Gameplay from Code

In its current version, Code Defenders is a code-based game. That is, playing it requires certain knowledge of Java and JUnit, because players interact with the system by explicitly editing and writing actual code. This not only reduces the number of potential players of the game, but also limits its playfulness. To overcome this narrowness of scope, we envision that fictional game scenarios can be designed in which mutation testing ideas are conveyed to the players without the actual need to write code. For example, previous work has shown that a software system can be visualised as a city and that this metaphor can facilitate certain program comprehension tasks [26], [27]. A similar approach could be taken to apply mutation testing concepts: if a city can represent the code under test, mutations can be mapped to some forms of attacks to the city, and unit tests can be mapped to defence elements which protect the city against the incoming attacks.

E. Empirical Evaluation

Recent work has shown that teaching mutation testing has the potential to improve the learning processes of novice students in programming courses [3]. The research hypotheses behind the development of CODE DEFENDERS is that by presenting mutation testing as a fun activity, a) players will produce strong tests and mutants for the classes involved in the gameplay, and b) players will ultimately perform better at testing tasks that can benefit from applying mutation testing, like fault localisation and test suite augmentation. Once CODE DEFENDERS reaches a higher level of maturity, we envision empirical studies to evaluate the validity of our hypotheses. A viable experimental setup to evaluate the impact on the technical skills of players would, for instance, require having a group of participants engage in the use of CODE DEFENDERS to learn and practice mutation testing after a tutorial session on the topic. Participants attending the same tutorial session but with no access to the game could be placed in a control group. All participants would be assigned a testing task where mutation testing skills are required, which will allow to assess whether or not the use of the game had any effect on their performance.

V. CONCLUSION

In this paper, following the trend of gamifying software engineering concepts, we advocate the gamification of mutation testing as a means to foster its adoption among software developers, which in turn might positively impact software quality in general. We have presented CODE DEFENDERS, an online game that aims to make mutation testing fun to learn and apply. CODE DEFENDERS maps the basic concepts of mutation testing into game elements, which include generating mutants, writing tests to detect those mutants, and also a protocol to deal with equivalent mutants as part of the gameplay. We have presented design and implementation details of the game, and have discussed some of the challenges we aim to tackle in future work. Meanwhile, for some mutation testing fun, CODE DEFENDERS is available for playing online at:

http://code-defenders.dcs.shef.ac.uk

ACKNOWLEDGEMENT

Thanks to Robert Sharp, Computer Science undergraduate at The University of Sheffield, for his contribution to the development of an earlier prototype of CODE DEFENDERS.

REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[2] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 654–665.

[3] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. S. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *Frontiers in Education Conference (FIE)*. IEEE, Oct 2015, pp. 1–9.

[4] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining "gamification"," in *International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek)*. ACM, 2011, pp. 9–15.

[5] K. M. Kapp, *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.

[6] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, "recaptcha: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

[7] L. von Ahn, "Duolingo: learn a language for free while helping to translate the web," in *International conference on Intelligent User Interfaces (IUI)*. ACM, 2013, pp. 1–2.

[8] L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," in *International Workshop on Games and Software Engineering (GAS)*. IEEE, 2012, pp. 5–8.

[9] N. Chen and S. Kim, "Puzzle-based automatic testing: bringing humans into the loop by solving puzzles," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2012, pp. 140–149.

[10] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability (STVR)*, vol. 7, no. 3, pp. 165–192, 1997.

[11] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 315–326.

[12] A. T. Acree, Jr., "On mutation," Ph.D. dissertation, Georgia Institute of Technology, 1980.

[13] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inf.*, vol. 18, pp. 31–45, 1982.

[14] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation Conference (GECCO)*. Springer Berlin Heidelberg, 2004, pp. 1338–1349.

[15] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability (STVR)*, vol. 23, no. 5, pp. 353–374, 2013.

[16] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 936–946. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818867

[17] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 919–930.

[18] O. Pedreira, F. Garca, N. Brisaboa, and M. Piattini, "Gamification in software engineering a systematic mapping," *Information and Software Technology (IST)*, vol. 57, pp. 157 – 168, 2015.

[19] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2014, pp. 37–42.

[20] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović, "Verification games: Making verification fun," in *Workshop on Formal Techniques for Java-like Programs (FTfJP)*. ACM, 2012, pp. 42–49.

[21] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, "Code hunt: Experience with coding contests at scale," *ACM/IEEE Int. Conference on Software Engineering (ICSE)(JSEET track)*, pp. 398–407, 2015.

[22] D. J. Dubois and G. Tamburrelli, "Understanding gamification mechanisms for software development," in *ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2013, pp. 659–662.

[23] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 433–436.

[24] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[25] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 2, pp. 276–291, 2013.

[26] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2007, pp. 92–99.

[27] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 551–560.