Teaching Software Testing Concepts Using a Mutation Testing Game

Benjamin S. Clegg[‡], José Miguel Rojas[*], Gordon Fraser[§]
Department of Computer Science, The University of Sheffield, Sheffield, United Kingdom
Email: {[‡]bsclegg1,[*]j.rojas, [§]gordon.fraser}@sheffield.ac.uk

***Abstract*—Software testing is a core aspect of software development, but testing programs systematically is not always a core aspect of software engineering education. As a result, software developers often treat testing as a liability, and overall software quality suffers. One of the reasons for this is that standard testing techniques are often perceived as boring and difficult when compared to creative programming and design activities, which dominate education. To make software testing education more enjoyable, we recently introduced the** CODE DEFENDERS **game, in which players engage with testing activities in a fun and competitive way. In this short paper, we explore the idea of using** CODE DEFENDERS **to systematically teach software testing concepts. We present a mapping of core developer testing concepts, such as statement or branch coverage, to categories of puzzles in the framework of the game. As players progress through levels of this game, they incrementally learn and practice testing concepts. By presenting software testing as an enjoyable activity, we hope that learners will not only acquire better testing skills, but will in the long term become better software engineers.**

***Keywords*-software engineering education; software testing; mutation testing;**

I. INTRODUCTION

If software is not systematically and thoroughly tested, then software quality suffers. Unfortunately, thorough testing is a difficult and error-prone task, and not every developer's favourite occupation. Recent results revealed that testing plays a much less prominent role in the daily activities of software developers than one would hope it to do [1], and frequent news reports on the effects of software bugs cast some doubt on the state of practice in software testing.

One important factor contributing to this situation is how developers are educated in software testing: Software engineering education generally tends to focus more on the creative aspects of design and coding. This is not surprising, as many standard testing techniques are laborious, and learning about them is much less engaging than writing new code.

In order to overcome this problem, we have recently introduced CODE DEFENDERS [6], a game where players compete over a snippet of code under test. Players can take the role of an *attacker*, who aims to break the program in such a way that this is not detected by the tests; a *defender* tries to produce the best possible tests to detect these attacks. The game concept is founded on principles of *mutation testing*, a testing technique where similar activities are performed by automated tools in order to evaluate the quality of tests and to guide generation of new tests. Both roles encourage and train an understanding of software bugs and how they are detected by tests, and how to create good tests.

In this short paper, we explore the idea of using this game concept to systematically teach core testing concepts. Whereas the previously presented multiplayer and duel modes [6] provide opportunity for learners to practice and hone their testing skills, the proposed systematic education approach is based on a single-player mode. Players progress through levels of increasing difficulty, each covering different testing concepts. Each level consists of a collection of individual testing puzzles, which are solved using either attacker or defender actions. Each puzzle, in turn, captures a particular testing concept by providing a fault that illustrates an insufficiency of a given test suite and requires the player to provide an additional test, or by asking the player to provide a fault that exposes the insufficiency of the given test suite.

We believe that by providing a comprehensive gamified curriculum of testing concepts captured in levels of the CODE DEFENDERS testing game, students will not only learn testing skills better; we also hope that they will perceive writing tests as a fun activity, and will thus become better software engineers who are more inclined to apply thorough testing.

II. THE CODE DEFENDERS GAME

The game concept underlying CODE DEFENDERS is mutation testing. This section introduces mutation testing, and how we have adapted it to the gameplay in CODE DEFENDERS.

A. Mutation Testing

Given a set of tests that pass on a program under test, mutation analysis [2] answers the question of how well the tests have exercised the program. Mutation operators are used to systematically create variants of the program under test, each differing by only a small syntactic change. The underlying premises are that software developers tend to write almost-correct programs (known as the *competent programmer hypothesis*) and that small syntactical changes can be representative of complex real-world program faults (known as the *coupling effect*). The tests are then executed on each of the mutants; if a test that passes on the original program fails on the mutant, then the mutant is *killed*; if no test kills a mutant, then the mutant is *live*. The mutation score is calculated as the ratio of live mutants to overall mutants, and provides a quantitative estimation of the thoroughness of the test suite. Live mutants can guide further test generation efforts. A particular aspect of mutation testing, and an inhibiting factor in practice, is that it is possible for a mutant to be semantically *equivalent* to the original program. In this case there exists no test that could distinguish it from the original program. Detecting equivalent mutants typically requires human intelligence.

B. Game Mechanics

CODE DEFENDERS integrates the main aspects of mutation testing (creating mutants, killing mutants, checking mutant equivalence) into a general gameplay framework. The gameplay consists of two player roles: *Attackers* aim to create mutants of the program under test that are as subtle as possible, making it as difficult as possible to reveal these changes with tests. Gameplay for attackers consists of editing the source code of the program under test directly. *Defenders* aim to create the strongest possible tests, such that they can detect the mutants created by the attackers. Gameplay for defenders consists of writing automated tests (e.g., JUnit). To ensure fair gameplay, both actions are subject to rules on possible modifications. For example, it is not possible to add new if-conditions or loops into a program, as that would make it easy to add very hard but meaningless mutants (e.g., checking for a very specific value).

The concept of checking mutants for equivalence is a core component of the game, since it is possible for attackers to create semantically equivalent mutants. This situation results in interaction between defender and attacker, where the defender challenges the attacker by claiming equivalence of a mutant, and the attacker then has prove non-equivalence by providing a test that kills the mutant, or accepts that the mutant is equivalent.

C. Game Modes

The basic gameplay consists of rounds of attack and defence (*duels*) of a single defender and an opposing attacker. The attacker takes the first turn of play. Once the attacker has submitted a mutant, it is the defender's turn to create a test with the right assertions to detect the mutants. A round is completed when the attacker has submitted a new mutant and the defender has countered with a new test. The game then proceeds until the number of rounds chosen when creating the game have been played. Upon completion of each round, a mutation analysis is performed to determine whether the newly created test has killed any live mutants. Defenders can choose to claim mutants as equivalent in each round, which then forces the attacker to handle the equivalence duel.

An alternative gameplay is based on asynchronous attacks and defences, where players can write tests or mutants without having to wait for the other player's turn to complete. This is particularly suitable to scale up to multiple participants in a *multiplayer* game; a team of attackers can compete with a team of defenders, thus providing additional incentive and adding possibilities for collaboration to the game.

A new gameplay mode introduced in this paper is the *single player* mode, which follows the concept of *duels*, but replaces the player's opponent with an automated player. When the player chooses the role of an attacker, then tests are generated by an automated tool such as EvoSuite [3]; when the player chooses the role of a defender, then mutants are generated by an automated mutation analysis tool such as Major [5]. In both scenarios, the level of difficulty can be influenced through the selection strategy of tests and mutants. For example, an easy mode could select randomly, while a more difficult mode could use coverage information.

D. Content and Information

The choice of content, i.e., which data to provide to the players, is an important aspect of any game. This aspect is particularly challenging in the case of software testing, as many metrics in testing are unreliable. For example, the number of tests on its own is not necessarily a sign of good testing, and code coverage is not necessarily related to fault finding effectiveness [4].

The main information provided to players is the code of the program under test, and information about which parts of the code are executed by tests. This is represented with highlighting in the source code representation, with the shade of the background colour of a line of code represents how frequently that line was executed by the tests. In addition, the source code is annotated with information about which lines were mutated. Players further receive statistics on the overall number of mutants and tests produced in a game so far.

Further information is provided depending on the game mode applied. In the *easy* mode, all players get to see all artefacts produced (i.e., tests and mutants). In *hard* mode, an attacker only gets to see details of the mutants produced, but not the actual tests produced; a defender only gets to see the tests produced so far, but not the actual mutants.

The idea of the hard mode is to balance the level of difficulty between the attacker and defender roles, and to increase the level of difficulty of the game and thus keep players engaged. However, in the context of a systematic educational tool without the competitive aspect of the game, we anticipate that full exposure of information (i.e., easy mode) may be preferable.

E. Game Incentives

CODE DEFENDERS builds on the competitive nature of the gameplay as a primary incentive to engage players. For this purpose, the game implements a point scoring system that aims to provide fair incentive to both, attackers and defenders, and to decide the winner of a game.

A mutant scores points for the attacker when it survives long in the game. The more rounds it survives, the more points the attacker scores. On the other hand, a test scores as many points for the defender as mutants it kills. In the case of equivalence duels, the attacker can score extra points by submitting a killing test for the mutant claimed equivalent, but can lose the points scored by that mutant otherwise. More details on the design, implementation and the point scoring system of CODE DEFENDERS can be found in an earlier publication on the gamified mutation testing system [6].

```java
int m(int x) {
  if (x == 5)
    return 3;
  return (x * 8);
}
```

(a) Code under test

```java
@Test
public void test() {
  assertEquals(3, m(5));
}
```

(b) Provided test

```java
int m(int x) {
  if (x == 5)
    return 3;
  return (x *3+7);
}
```

(c) Mutant

```java
@Test
public void test() {
  int x = 3;
  assertEquals(x*8, m(x));
}
```

(d) Expected test

Fig. 1: Statement coverage example. Fig. 1a shows the original method; Fig. 1b shows the unit test provided for the attacking role; Fig. 1c shows a example of mutant not killed by the existing test; and Fig. 1d shows a unit test that kills the mutant.

III. Code Defenders as an Education Platform

A. *Implementing a Testing Curriculum with* Code Defenders

To use Code Defenders for systematically teaching testing concepts, we propose to take the following approach:

- For each individual testing concept that should be covered there is a *level* in Code Defenders.
- Each level is a collection of *games* of increasing difficulty.
- Each game consists of a program under test together with a test suite, which is lacking at least one test that is required in order to suitably test the program under test with respect to the testing concept underlying the level.
- Games can alternate between attacker and defender scenarios. When playing a game as a defender, then a pre-defined mutant that is not captured by the existing test suite illustrates a deficiency, which requires additional tests based on the underlying testing concept. When playing a game as an attacker, the aim is to find a mutant that is not discovered by the existing test suite, because it does not satisfy the testing concept underlying the level.
- The scoring system would need adaptation: Scores accumulate across games, and the score for each game could reflect the number of attempts and quality of solution.
- Equivalence duels can be used at the end of a level as a means of assessment. In order to decide equivalence, the testing concept needs to be suitably applied; the resulting test suite will support the decision about equivalence.

B. *Testing Concept Examples*

In this section, we explore a number of different testing concepts and demonstrate how they can be mapped to games in the Code Defenders framework.

1) Statement Coverage: In order to discover or confirm bugs in programs, their code must be executed. Intuitively, if a statement containing a bug is never executed by any test, then the bug cannot be detected. Statement coverage, hence, is the most general criteria to measure test quality and understanding it is fundamental for any software engineer.

```java
int m(int x) {
  int y = 1;
  if (x == 5) {
    y = 3;
  }
  return (y * 8);
}
```

(a) Code under test

```java
@Test
public void test() {
   assertEquals(24, m(5));
}
```

(b) Provided test

```java
int m(int x) {
  int y = 0;
  if (x == 5) {
    y = 3;
  }
  return (y * 8);
}
```

(c) Mutant

```java
@Test
public void test() {
  int x = 3;
  assertEquals(8, m(x));
}
```

(d) Expected test

Fig. 2: Branch coverage example

Figure 1 shows an example game that teaches statement coverage: Figure 1a shows the program under test, and Figure 1b is the initial test suite. In this example, only the true branch of the if-condition is covered, while the second return-statement is not covered. This means that the mutant shown in Figure 1c is not detected by this test suite, requiring an additional test (Figure 1d) to fully cover all statements of the program. When played as an attacker, the player would be shown the original program and test suite, and the task would be to generate a mutant that is not detected by the test suite. To ensure the mutant is non-equivalent, each game would further require a golden test suite that is mutation-adequate, or a test generation tool that automatically performs this check. When played as a defender, the player would be shown the original program, test suite, and mutant, and the task would be to write a test guided by the target concept that detects the mutant.

2) Branch Coverage: Branch coverage requires that all edges in the control flow graph of a program are executed at least once, which implies that each if-condition needs to evaluate to true and to false. Figure 2 shows an example game covering the concept of branch coverage. The initial test suite (Figure 2b) covers all statements, but not all branches. Thus, a mutant like Figure 2c remains undetected, and provides the potential for playing as attacker or defender.

3) Testing Loops: Various coverage criteria have been defined to deal with the challenges arising from loops. For example, different numbers of loop iterations may lead to discovery of different faults. Figure 3 shows an example game where the existing test suite covers the case where the loop is not executed and executed exactly once, but it does not cover the case where the loop is covered multiple times (i.e., it does not satisfy the boundary-interior coverage criterion). Again this enables a mutant that is not discovered by the existing test suite and provides the basis for a game.

4) Boundary Value Testing: Figure 4 shows an example where the existing test suite achieves branch coverage, but does not test at the boundary of $x \geq 100$. Solving this game requires either generating a mutant on the boundary value, or creating a test that detects a mutant on the boundary value.

```
int m(int x) {
  int r = 1;
  for (int i=0; i<x; i++){
    r++;
  }
  return r;
}
```

(a) Code under test

```
@Test
public void test0() {
  assertEquals(1, m(0));
}
@Test
public void test1() {
  assertEquals(2, m(1));
}
```

(b) Provided test

```
int m(int x) {
  int r = 1;
  for (int i=0; i<x; i++){
    r= r*2;
  }
  return r;
}
```

(c) Mutant

```
@Test
public void test() {
  int x = 3;
  assertEquals(3, m(2));
}
```

(d) Expected test

Fig. 3: Loop testing example

```
int m(int x) {
  if(x > 100)
    return +1;
  else
    return -1;
}
```

(a) Code under test

```
@Test
public void test0() {
  assertEquals(1, m(200));
}
@Test
public void test1() {
  assertEquals(-1, m(0));
}
```

(b) Provided test

```
int m(int x) {
  if(x >= 100)
    return +1;
  else
    return -1;
}
```

(c) Mutant

```
@Test
public void test() {
  int x = 100;
  assertEquals(-1, m(x));
}
```

(d) Expected test

Fig. 4: Boundary value analysis example

```
int m(int x) {
  int y = 0;
  if(x > 100) {
    y = 1;
  }
  if(x > 1000) {
    y = -1;
  }
  return x*y;
}
```

(a) Code under test

```
@Test
public void test0() {
  assertEquals(-1001, m(1001));
}

@Test
public void test1() {
  assertEquals(0, m(0));
}
```

(b) Provided test

```
int m(int x) {
  int y = 0;
  if(x > 100) {
    y = -1;
  }
  if(x > 1000) {
    y = -1;
  }
  return x*y;
}
```

(c) Mutant

```
@Test
public void test() {
  int x = 500;
  assertEquals(500, m(x));
}
```

(d) Expected test

Fig. 5: Dataflow testing example

5) Dataflow Testing: Based on the insight that software errors are often the result of incorrect variable assignments, dataflow testing measures how well interactions between variables are tested. It relies on the notion of a *definition*, when a value is stored in a variable, and *use*, when the value of a variable is read. Figure 5 shows an example game where branch coverage is achieved by the existing test suite, but only a def-use adequate test suite will detect a mutant of the assignment to y in the first if-condition.

6) Mutation Testing: As CODE DEFENDERS is built on mutation testing, it is a natural fit for this testing concept.

7) Model-based Testing: All the testing concepts mentioned above are based on code. However, there is nothing that would preclude the application of CODE DEFENDERS to models instead. For example, concepts like coverage of a finite state machine (FSM) or conformance testing could be implemented by having players mutate and test FSM models.

8) Combinatorial Testing: It will also be possible to cover generic concepts such as combinatorial testing. For example, a test input could be given in terms of a combination of values related to a GUI component, rather than raw JUnit code.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed the use of the CODE DEFENDERS game concept to develop systematic exercises that teach a range of different testing concepts. Each concept is implemented as a series of individual games on small programs under test, and players have to either improve a test suite to reveal a mutant that depends on a target testing concept, or produce a mutant that exploits the absence of the target testing concept in an existing test suite. The notion of equivalence duels could serve as an assessment mechanism in this context.

The challenge ahead now is to produce a suitable number of games that exercise the testing concepts listed in this paper, as well as any other applicable concepts. Ideally, these examples would be of increasing difficulty, allowing a smooth progression of the learner without leading to confusion (too challenging) or frustration (too easy). An overarching narrative may help to engage students. Once a testing curriculum is implemented using these games, a further challenge lies in evaluating the effects these have on learners.

CODE DEFENDERS is available to play online at http://code-defenders.org and is made public as an open-source project at http://github.com/jmrojas/codedefenders.

REFERENCES

[1] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[3] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, 2013.

[4] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 435–445. ACM, 2014.

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

[6] J. M. Rojas and G. Fraser. Code Defenders: A Mutation Testing Game. In *Int. Workshop on Mutation Analysis*, pages 162–167. IEEE, 2016.