Adaptation of an Online Platform to Teach Testing

Lydie du Bousquet[1,*], Christophe Saint-Marcel[1]

[1]*Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble*

Abstract

Software quality is seen as an integral part of Software Engineering education. A key concept is software testing, the primary method used in the industry to evaluate the software quality. In this article, we focus on software testing assessment. We present 3 contributions: (1) the adaptation of our online learning open platform (Caseine) for software testing assessment, (2) an approach to create testing exercises, and (3) a (still) modest set of exercises (but currently under extension).

Keywords

Software testing, Mutation analysis, Assessment method, Educational tool

1. Introduction

At *Université Grenoble Alpes*, France, we have been proposing a course on software testing at the master level for more than 20 years. The main objective of this course is to improve the skills of students to generate test cases. The course is composed of 4 sessions of 3 hours. It includes a classical teaching approach based on lessons, exercises relying on printed code and exercise corrections on a blackboard.

For a long time, we have been seeking a solution to introduce more practical work (1), compliant with remote learning (2), that includes automatic grading (3) to check that the different testing strategies are correctly applied by the students (4). The Covid pandemic decided us to to search for a solution that allows to automatically evaluate if our students properly applied a given test strategy to produce an executable test suite (research question).

This article presents three contributions. The first one is the adaptation of Caseine, an online learning platform developed by the Université Grenoble-Alpes (UGA), originally dedicated to programming learning. The second one is an approach to create testing exercises. The final one is a (still) modest set of exercises (but currently under extension), that are shared with the community on Caseine in an open course[1].

In the following, section 2 gives some elements about classical methods to evaluate test quality. Section 3 explores different tools proposed by other universities for teaching how to test. Sections 4 and 5 present Caseine and its adaptation for our pedagogical purpose. Section 6 details our approach for creating our testing exercises. Sections 7 and 8 provide feedback, conclusion and perspectives.

6th Software Engineering Education Workshop (SEED-2022)

*Corresponding author.

lydie.du-bousquet@univ-grenoble-alpes.fr (L. du Bousquet); christophe.saintmarcel@velossity.fr (C. Saint-Marcel)

[1]https://moodle.caseine.org/course/view.php?id=825

2. Software testing and quality

Software Testing is the process of executing a program or a system with the intent of finding errors [1]. It is usually considered to be the primary method used in the industry to evaluate software quality [2]. Basically, software testers must select a relevant set of test cases, execute it, check if the system under test behaves as expected, and decide when the process is over.

To select test cases, a software tester can use several strategies as proposed in the literature. A testing strategy typically relies on an abstract model to be covered, which can be a graph, a logical expression, an input domain partition, or a syntactic description [2, 3, 4]. Several tools allow checking that test suites reach dedicated coverage criteria, such as statement or branch coverage, but they do not assess that the tests can find errors. Until now, to evaluate the ability of a test set to find errors, two main strategies were proposed: mutational analysis and fault injection.

Mutational analysis relies on a set of elementary mutation operators, which are applied *systematically* on the program under test to produce a set of faulty versions, called *mutants* [5]. For example, a '+' is replaced by a '−', a '<' is replaced by a '>' or a '≥' (etc.). Tests are executed against all the mutants. Each time a fault is detected, the related mutant is marked as *killed*. The proportion of killed mutants denotes the *mutation score*. The higher the score, the better the test suite is.

Fault injection is a strategy close to the previous one, also based on the production of erroneous programs. In this approach, the chosen faults are usually those already identified during the development. The major difference with the mutation analysis relies upon a non-systematized production of erroneous programs.

Both approaches have some weaknesses, but one usually considers that they assess reasonably the quality of test suites. For this reason, they are both commonly used for assessing software testing teaching, as detailed in the next section [6, 7, 8, 9].

3. Teaching testing

In [10], authors report that testing teachers face different problems amongst which the lack of tools for teaching support, the lack of practical examples available. However, several initiatives exist.

PABS (for Programming Assignment Feedback System) is a tool that automatically generates feedback for programming assignments and provide an overview over related work (e.g. functional tests and style checks) [9]. The whole system is based on SVN. The evaluation of tests is based on different criteria, among which a coverage evaluation.

In [6], authors present their approach (based on case studies) and their teaching material. The students' tests are evaluated through code coverage and mutation analysis. They do not detail their platform.

VU-BugZoo is a platform to teach software testing, based on a repository of faulty (standalone and embedded) code. Students are engaged in an "bug-hunting" experience [7].

Code Defenders is a game proposed to teach software testing concepts [8]. The player can be either an attacker or an defender. The attacker introduces fault in Java programs in such a way that it is not detected by the tests. The defender produces JUnit tests to detect these attacks. The game is founded on principles of mutation testing. *Code Defenders* is open-source and available on GitHub [11].

ProgTest is a web-based tool for the submission and automatic evaluation of programming assignments based on testing activities [12]. Teachers need to provide a correct program and an adequate test set. The environment integrates testing tools such as JUnit, for Java. Tests of students are evaluated with a coverage tool and a mutation testing tool (JABUTI and JUMBLE for Java).

We wanted a solution to automatically assess the test cases (like all the previous tools), in which teachers can choose the mutants they want to evaluate (different from *ProgTest*), in order to check the good application of different testing strategies (with the same idea than *Code Defenders* but not necessarily linked to classical mutation operators). Moreover, we wanted our solution to be integrated with tools that are used in our university, in order to limit the number of pedagogical environments the students are faced with. For this reason, we chose to work within the Caseine environment, detailed hereafter.

4. Caseine environment

Caseine[2] is an online learning platform developed by the Université Grenoble-Alpes (UGA). It is based on Moodle *Virtual Programming Labs* (VPL), activity modules that manage programming assignments based on input and output cases[3] [13]. Many plugins are available in Moodle. In particular, there exist VPL for more than 30 programming languages.

Compared to a classic Moodle instance, Caseine is original in that the environment is open to all universities in the world (or almost) via the federation of Identities Education Research Edugain which contains, for example, Renater. Moreover it allows the sharing of resources and many plugins are available (a large community of teachers which goes far beyond Grenoble contributes to the expansion of content and of the platform itself). Moreover, Caseine proposes additional facilities to the classic Moodle, amongst which:

- different types of test formats: classical Moodle VPL one (based on input/output couple) and usual test framework (such as *JUnit* for Java, or *unittest* (formerly *PyUnit*) for Python); and
- links to classical IDEs, which allow students to work offline, on their personal computer with their favorite development environment.

In Grenoble, Caseine is mainly used for programming teaching. The students work in a classroom or remotely. They complete and execute the given programs using the Caseine web interface (Fig.1(a)) or their local IDE (e.g. Eclipse). At the end of the exercise, the students have to ask for an evaluation of their production (Fig.1(b)). Programs are uploaded on Caseine if needed, and tests provided by the teachers are executed. A score is immediately returned ((Fig.1(c)). It scores maximum if all tests pass. It decreases with the number of fail tests.

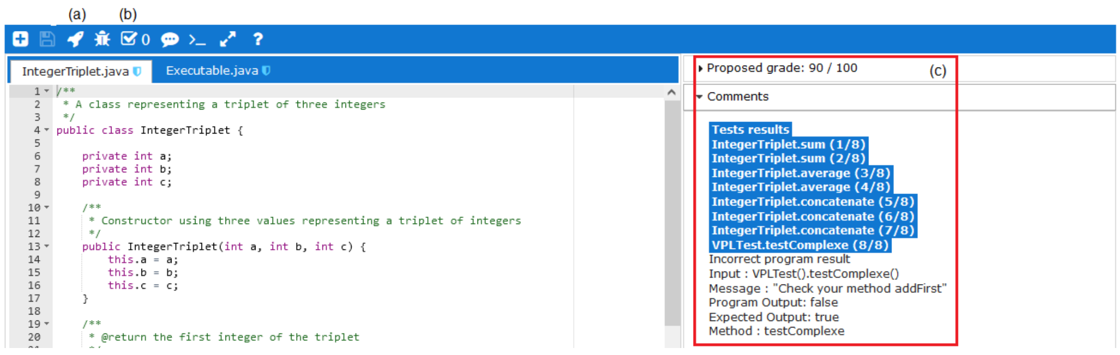
5. Caseine adaptation for tests assessment

The evaluation process in Caseine follows a natural programming viewpoint: the expected result of a programming activity is to produce a correct program, i.e. a program for which all tests pass.

To evaluate the quality of tests, we needed to change the viewpoint. Indeed, since tests should be designed to find the errors [1], a software tester is "satisfied" when his tests provoke failures: it means that "his/her tests are doing their job".

We wanted to be able to evaluate the ability of the tests designed by students to provoke failure. The first

[2]https://moodle.caseine.org/
[3]https://moodle.org/plugins/mod_vpl



```
/**
 * A class representing a triplet of three integers
 */
public class IntegerTriplet {

    private int a;
    private int b;
    private int c;

    /**
     * Constructor using three values representing a triplet of integers
     */
    public IntegerTriplet(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    /**
     * @return the first integer of the triplet
```

```
Proposed grade: 90 / 100
Comments

Tests results
IntegerTriplet.sum (1/8)
IntegerTriplet.sum (2/8)
IntegerTriplet.average (3/8)
IntegerTriplet.average (4/8)
IntegerTriplet.concatenate (5/8)
IntegerTriplet.concatenate (6/8)
IntegerTriplet.concatenate (7/8)
VPLTest.testComplexe (8/8)
Incorrect program result
Input : VPLTest().testComplexe()
Message : "Check your method addFirst"
Program Output: false
Expected Output: true
Method : testComplexe
```

Figure 1: Student's view of a programming exercise in Caseine

step consisted in the creation of a specific kind of VPL able to evaluate the tests of the students, with a classical mutation analysis/fault injection approach.

Using this new type of VPL, teachers define the test file(s) to be completed by students and a set of faulty programs (e.g. mutants) to evaluate the quality of the students' production. Note that the teacher can build the mutants the way he/she wants: with fault injection or mutation analysis, manually or with a tool.

Achieving a programming exercise or a testing exercise is transparent for the students: the process and the interfaces are the same. During the evaluation phase for a testing exercise, the student's tests are executed against the mutants. The student's assessment is proportional to the mutation score: it is maximum if the students' tests kill all the mutants.

The creation of a VPL type dedicated to testing in Caseine provides a mean to automate the evaluation of the students tests. This is our **first contribution**. In the next section, we detail how we used it to check that the students apply properly the different testing approaches.

6. Testing process evaluation

To check that students properly apply test generation strategies, we **adapt** the mutation analysis approach. This is our **second contribution**.

In software testing theory, a test generation strategy relies on a coverage criterion built upon an artefact such as the specification, the code, the input domain, or a fault model [14]. Given an artefact A, a test criterion C_E will be expressed as a set of n elements $E = \{e_k, 1 \leq k \leq n\}$ to cover.

Let $\mathrm{cov}(e, p, i)$ be a Boolean function that returns true when the execution of program p with an input $i \in I$ covers an element $e \in E$. In the following and without losing generality, we suppose that I denotes only the set of possible inputs of p, as well as E contains only reachable elements, i.e. $\forall k, 1 \leq k \leq n, \exists i \in I | \mathrm{cov}(e, p, i)$.

To evaluate the correct application of a test criterion C_E, we build n mutants, each of them corresponding to one element to cover. To do that, we proposed a transformation strategy for each test criterion C_E we study during our course.

Let TR_{C_E} be such a transformation strategy. Considering a program p and an element e_k to be covered, $TR_{C_E}(p, e_k) = M_k$ defines a family of mutants, such that each mutant $m_k \in M_k$ will produce a different output than the original program p whenever an input i covers the element e_k:

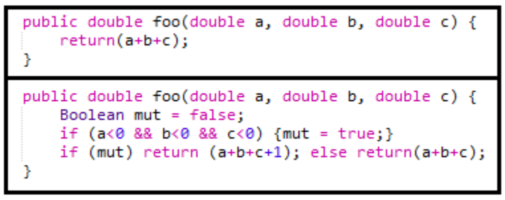
$$\forall m_k \in M_k, \forall i \in I, \mathrm{cov}(e_k, p, i) \iff p(i) \neq m_k(i)$$

A transformation strategy is similar to the original mutation operators since it is applied in a systematic way. It is different in that the modifications in the program under test are "larger" than the syntactic variation classically defined by the mutation operators.

Informally, each mutant is built on 3 kinds of modifications:

1. New variables are introduced, among which a Boolean one called *mut*, initialized to `false`.
2. Statements are introduced in the code to compute whether element e_k is covered. When it is the case, *mut* variable is set to `true`.
3. A conditional statement is introduced to alter of the result depending on the value of *mut* variable. Since the result can be altered in many ways, a set of mutants can be produced. Only one is necessary for each element e_k.

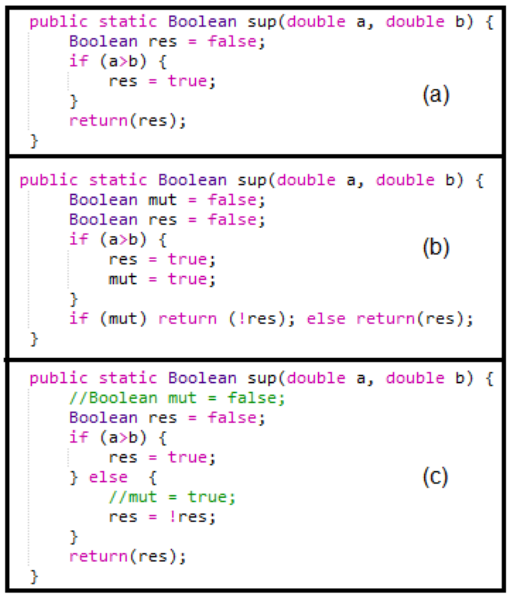
In the following, we illustrate the strategies on different criteria.



```
public double foo(double a, double b, double c) {
    return(a+b+c);
}
```

```
public double foo(double a, double b, double c) {
    Boolean mut = false;
    if (a<0 && b<0 && c<0) {mut = true;}
    if (mut) return (a+b+c+1); else return(a+b+c);
}
```

Figure 2: An example of fault to detect the partition $(a<0)$ && $(b<0)$ && $(c<0)$



```
public static Boolean sup(double a, double b) {
    Boolean res = false;
    if (a>b) {
        res = true;                                (a)
    }
    return(res);
}
```

```
public static Boolean sup(double a, double b) {
    Boolean mut = false;
    Boolean res = false;
    if (a>b) {                                     (b)
        res = true;
        mut = true;
    }
    if (mut) return (!res); else return(res);
}
```

```
public static Boolean sup(double a, double b) {
    //Boolean mut = false;
    Boolean res = false;
    if (a>b) {
        res = true;
    } else  {                                      (c)
        //mut = true;
        res = !res;
    }
    return(res);
}
```

Figure 3: An example of faults for block coverage
(a) Original program
(b) Mutant required to assess statement coverage
(c) Additional mutant to assess branch coverage

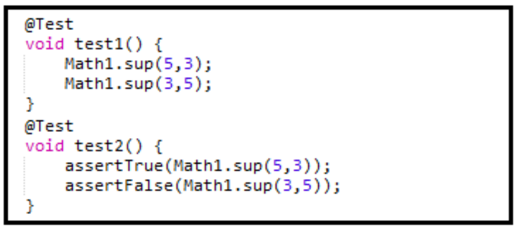
6.1. Category Partition testing

This testing method guides the tester to define a partition of the input space $I = \{\pi_1, \pi_2, ..., \pi_m\}$. The implicit hypothesis is that the inputs belonging to a partition are equivalent to detect the faults, so the tester has to choose only one input in each partition [15].

To check if the strategy is well applied, we propose exercises in which the categories and the constraints are explicitly given. The students have to produce a test suite that covers the partitions ($E = \{\pi_i, \pi_2, ..., \pi_m\}$).

Thus we build one mutant for each partition on the model given Fig.2: the variable *mut* is set at the beginning of the program to `true` if the input variables combination corresponds to a partition.

The same idea is used to evaluate pairwise coverage [14]. This strategy is usually applied to select a subset of parameter combinations, so that each pair of parameters (resp. categories) has to be covered. The resulting set of combinations is much smaller than the exhaustive one, because each instantiation of the parameters covers several expected pairs. In this case, the *mut* variable is set to `true` at the beginning of the program if the inputs correspond to the pair to be covered.



```
@Test
void test1() {
    Math1.sup(5,3);
    Math1.sup(3,5);
}
@Test
void test2() {
    assertTrue(Math1.sup(5,3));
    assertFalse(Math1.sup(3,5));
}
```

Figure 4: Example of tests for Sup function

6.2. Statement and Branch coverage

Both methods are code-based testing methods. They require to cover a set of blocks (corresponding to the code branches). A mutant is thus produced for each block by setting *mut* to `true` in the block to be covered (Fig. 3(b)). Note that it is necessary to create an *else* bock in a mutant when it does not exist in the original code for branch coverage (Fig. 3(c)). Moreover, the modifications in the code can sometimes be simplified. In Fig. 3(c), we get rid of the *mut* variable and directly modify the result in the *else* branch.

Using a mutation approach to evaluate coverage might sounds strange since there are plenty of test coverage tools. However, those tools do not assert the quality of the test oracles (the mechanism to determine whether a test has passed or failed). Fig 4, the first test achieves branch coverage but does not check the expected results. It cannot detect faults in contrary to the second one.

6.3. Transition coverage

When the specification is expressed as an automaton, it is suggested to produce tests in order to cover states, transitions, or pairs of transitions. Modification of the code increases with the complexity of the coverage criterion.

In Fig. 5(a), we provide a specification example of a turnstile. We have thus $E_t = \{t_i \mid 0 \leq i \leq 4\}$ for transition coverage. Fig 5(b) gives an example of mutant to achieve transition coverage: transitions are detected in the code and result is altered in consequence.

For the same example, for pairs of transitions coverage, we have $E_{t^2} = \{(t_0, t_1), (t_0, t_4), (t_1, t_2), (t_1, t_3), (t_2, t_2), (t_2, t_3), (t_3, t_1), (t_3, t_4), (t_4, t_4), (t_4, t_1)\}$. To produce a mutant to evaluate pair of transitions coverage, we need to add a global variable to memorize the previous transition. The variable is updated each time a transition is followed, which induces an important code instrumentation to compute the “previous transition” (variable `prevTrans` Fig 6). However, this instrumentation is the same from all the mutants related to the criterion.

7. Feedback

We have used Caseine for testing for two years now. Four groups of students in Master 2 Computer sciences and one group of students in Master 1 computer sciences experience different exercises. We did not conduct formal evaluation, but we collected their informal feedback. It shows that they were happy to have an opportunity to apply concretely the testing approaches. A formal evaluation is planned in November 2022, with a new group of Master 2.

From the teacher point of view, the possibility to share exercise was appreciated: two other teachers decided to use Caseine for their testing course.

8. Conclusion and perspectives

In this article, we explain how we adapt Caseine, an online environment to teach programming, in order to teach testing. We also show how the mutation-based approach can be adapted to evaluate if the different testing approaches are correctly applied by the students. This method was suggested in [14].

Currently we have prepared 10 exercises (Fig. 7). Six of them were previously tested by 3 classes of students and 4 will be used in November 2022. They cover the different strategies as well as a classical mutation approach. We proposed an open course on Caseine[4], in which the validated exercises are available.

The whole approach has some advantages and weaknesses. The advantages are largely tied to the Caseine platform:

- everyone can ask for a connection to Caseine,
- the exercises can be shared,
- the students can work online or with their usual development environment,
- they have an immediate feedback related to their tests in the same way as they have feedback about their programs,
- the teachers can follow the students progress in a easy way and prevent dropout,
- they can use classical tools to produce their mutants,

[4]https://moodle.caseine.org/course/view.php?id=825

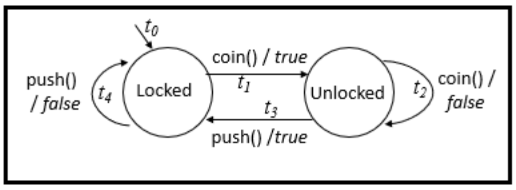


Figure 5: Turnstile specification

The strategy to produce the mutants is not yet automated. It might be difficult to automate for a specification-based approach, unless the code is generated automatically from the specification. For code-base testing strategies, the work of CEA LIST on the production of labels to capture different notions of test coverage shows that part of the process could be automated [16].

The set of available exercises is still modest, but we are currently in the process of developing it, in order to propose exercises from classical text books.

Acknowledgments

Thanks to Sumaiya Sultana, Al-Bashir Muhammad et Mpoki Mwaisela who contributed to extend the exercises within the project “région Auvergne Rhône-Alpes RAVEN” (PAI 21 007381 01). The Caseine environment was partially funded by the “Université Grenoble-Alpes Idex” (ANR-15-IDEX-0002), the “PERSYVAL-Lab LabEx” (ANR-11-LABX-0025-01), and “Flexi-TLV”, three “Investissement d’Avenir” programs.

References

[1] G. J. Myers, The Art of Software Testing, Wiley, 1979.

[2] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, USA, 2008.

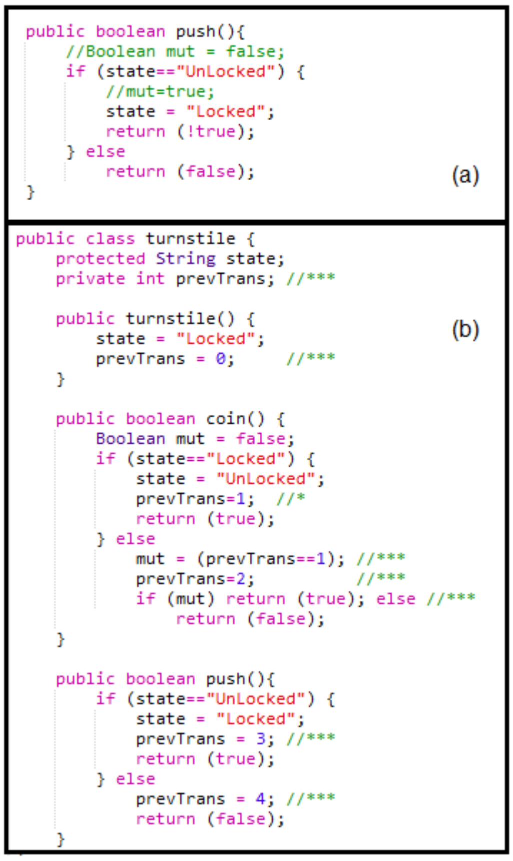
[3] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 1983.

[4] R. V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.

[5] R. DeMillo, R. Lipton, F. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (1978) 34–41.

[6] F. Dadeau, J.-P. Gros, F. Peureux, A case-based approach for introducing testing tools and principles, in: IEEE Int. Conf. on Software Testing, Verif. and Validation Workshops (ICSTW), 2020, pp. 429–436.

[7] N. Silvis-Cividjian, R. Limburg, N. Althuisius, E. Apostolov, V. Bonev, R. Jansma, G. Visser, M. Went, Vu-bugzoo: A persuasive platform for teaching software testing, in: ACM Conf. on Inno-



```
public boolean push(){
    //Boolean mut = false;
    if (state=="UnLocked") {
        //mut=true;
        state = "Locked";
        return (!true);
    } else
        return (false);
}
```

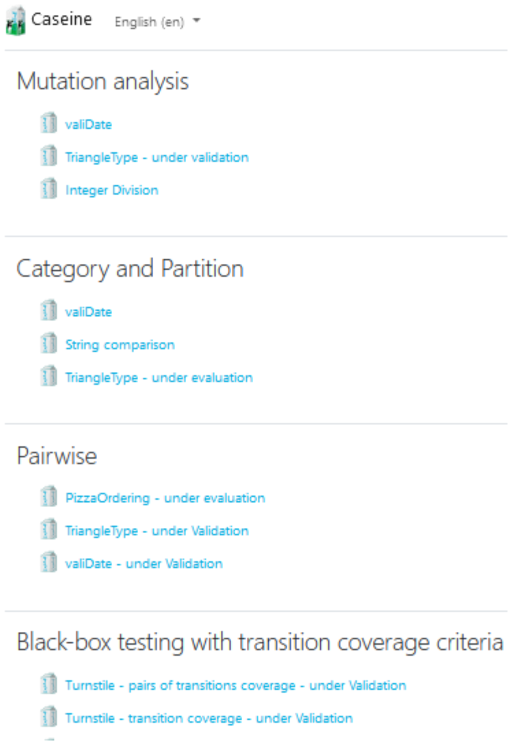
```
public class turnstile {
    protected String state;
    private int prevTrans; //***

    public turnstile() {
        state = "Locked";
        prevTrans = 0;      //***
    }

    public boolean coin() {
        Boolean mut = false;
        if (state=="Locked") {
            state = "UnLocked";
            prevTrans=1;  //*
            return (true);
        } else
            mut = (prevTrans==1); //***
            prevTrans=2;          //***
            if (mut) return (true); else //***
                return (false);
    }

    public boolean push(){
        if (state=="UnLocked") {
            state = "Locked";
            prevTrans = 3; //***
            return (true);
        } else
            prevTrans = 4; //***
            return (false);
    }
```

Figure 6: Two mutants generated for Turnstile
(a) Mutant to detect t_1
(b) Mutant to detect (t_1, t_2)



Mutation analysis

- valiDate
- TriangleType - under validation
- Integer Division

Category and Partition

- valiDate
- String comparison
- TriangleType - under evaluation

Pairwise

- PizzaOrdering - under evaluation
- TriangleType - under Validation
- valiDate - under Validation

Black-box testing with transition coverage criteria

- Turnstile - pairs of transitions coverage - under Validation
- Turnstile - transition coverage - under Validation

Figure 7: Shared exercises on Caseine

vation and Technology in Computer Science Education (ITiCSE), 2020, p. 553.

[8] J. M. Rojas, G. Fraser, Code defenders: A mutation testing game, in: IEEE 9th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), 2016, pp. 162–167.

[9] L. Beierlieb, L. Iffländer, T. Schneider, T. Prantl, S. Kounev, Teaching software testing using automated grading, in: 5th Workshop Automatische Bewertung von Programmieraufgaben (ABP), 2021.

[10] S. M. Melo, V. X. S. Moreira, L. N. Paschoal, S. R. S. Souza, Testing education: A survey on a global scale, in: XXXIV Brazilian Symp. on Software Engineering (SBES), 2020, p. 554–563.

[11] G. Fraser, A. Gambi, J. M. Rojas, Teaching software testing with the code defenders testing game: Experiences and improvements, in: IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), 2020, pp. 461–464.

[12] D. M. de Souza, B. H. Oliveira, J. C. Maldonado, S. R. S. Souza, E. F. Barbosa, Towards the use of an automatic assessment system in the teaching of software testing, in: IEEE Frontiers in Education Conference (FIE) Proceedings, 2014, pp. 1–8.

[13] J. C. Rodríguez-del Pino, E. Rubio Royo, Z. Hernandez Figueroa, A virtual programming lab for moodle with automatic assessment and anti-plagiarism features, in: Int. Conf. on e-Learning eBusiness Enterprise Information Systems & e-Government, 2012.

[14] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2016.

[15] S. K. Khalsa, Y. Labiche, Extending category partition's base choice criterion to better support constraints, Journal of Software: Evolution and Process 30 (2018).

[16] N. Kosmatov, Combinations of Analysis Techniques for Sound and Efficient Software Verification, Hdr, U. de Paris-Sud 11, France, 2018.