A Software Framework Based on JUnit for Automated Software Testing in Computer Science Courses

Michael A. Zmuda
Dept. of Computer Science and Software Engineering
Miami University
Oxford, OH USA
zmudam@miamioh.edu

***Abstract*—This innovative practice full paper describes a new software framework based on JUnit to test student work. Automated testing is an important capability when teaching software development at the college level. Ideally, a testing system will allow the instructor to efficiently create a thorough set of tests. Also, the software should facilitate grading tasks and produce informative reports that can be distributed to the students in a timely fashion. For Java development, the well-known JUnit framework enables a test suite to be applied to a student's submission. The mutools library presented here extends the JUnit framework in novel ways to accelerate the instructor's task of creating test suites. This new framework allows the instructor to augment tests with directives to control scoring and reporting. The four main capabilities of the software include: 1) An assert statement that does not terminate the test when it has failed. Instead, statistics are maintained regarding the success or failure of each assert statement. 2) Tests that can be configured to award partial credit. This can be useful in situations where the instructor deems it appropriate to award students some credit even in the presence of incorrect asserts. 3) Tests that can be grouped into categories that match a particular rubric item. Java annotations are placed on the test suite to define these categories. For example, @TestCategory(name="remove", points=10.0) specifies that 10 points will be awarded for successful implementation of all tests related to removing an item from the collection class. 4) Testing reports that contain varying levels of information. With minimal details, the testing report shows each testing category with the following information: assert statistics, whether the test timed out or had abnormal termination. This software has been used for many semesters and has been found to increase the speed at which the instructor can develop test suites for grading. The framework is available on github.**

Keywords—automated grading, educational software, grading systems

I. INTRODUCTION

Within computer science education, automated assessment (AA) has been an important area of research for many years. The overarching goal of this body of work is to automate the process of providing students with meaningful feedback in an efficient manner. This capability is important because class sizes are often very large and the number of assignments given in a semester can be numerous. In these situations, it is costly for instructors and teaching assistants to manually provide individualized instruction or perform individualized grading. Even with unlimited manpower, there still may be advantages to automating the process. Automation provides more consistent and correct feedback to the students and can improve interactions with TAs [1], [2].

The AA field contains many different tools addressing many different objectives. Some of the tools are interactive in nature and provide real-time instruction. Some tools inform the student about the presence of errors and others assist students in fixing their errors. This current work focuses on the task of identifying errors in student code, which is still a commonly performed task. Within this domain, many different types of student work exist. For example, the student work can be: a traditional programming task, a GUI application, a system administration task, web application, or a mobile application. Each of these has its own unique challenges. This paper focuses on identifying errors within traditional programming assignments.

II. RELATED WORK

[3] and [4] both performed exhaustive literature reviews on AA research. The existing techniques were classified along various dimensions. Although they differ, each review identifies key attributes and are largely aligned. A subset of the core classifications identified by these two reviews are used here to provide context for the current work.

There are different types of feedback produced by AA tools. Tools can provide students hints on how to approach a problem. Some tools provide help on how to improve a solution or correct a bug. For recursive functions, for example, the student may be reminded about the base case and recursive step. The most pervasive category is providing feedback on mistakes such as: test failures, compile errors, stylistic problems [5], code quality, and performance issues. All of these forms of feedback may be relevant to a single assignment. Since a single tool does not necessarily address all of these issues, several different tools may be utilized for a single assignment. The current work fits within the *test fai*lure category.

Within the *test failure* category, several papers have used JUnit [6] as the underlying unit testing framework for Java code, as does the current work presented here. JUnit++ [7] extends JUnit for reasons like the current work. Namely, using JUnit alone for grading is "cumbersome, repetitive, and error prone." [7] The purpose of JUnit++ is to test for correctness and assess functional properties of the submission using Java's

reflection capabilities. [8] developed AutoGrader to process multiple student submissions through an instructor-developed JUnit test suite. AutoGrader also assesses coding style. COALA [9] provides advice to students by applying fuzzy logic to software metrics and test cases. The use of JUnit within Eclipse displays the default pass/fail results. Other popular languages have their own unit testing frameworks and the techniques described here may be applicable to those frameworks. To be sure, developing good test suites is challenging. [10] and [11] both provide approaches that yielded results better than traditional testing.

The techniques used to produce the feedback are another important consideration. Static and dynamic code analysis can be used to analyze code and provide feedback. The *adaptability* of the tool is considered high if the tool can be easily configured for new exercises. Providing test cases and solutions are two approaches that allow instructors to adapt to new assignments. The current work falls into the *test case* category.

III. Proposed Solution

The JUnit software framework is a framework for testing software. It is well known in the software community and can be run from the command line, which promotes automated testing. JUnit's target audience is professional software developers, not computer science instructors. As such, some of the tasks performed by instructors are not directly supported by JUnit. For example, JUnit does not provide any support for computing scores based on test results. Also, tests that fail softly are not possible. And, the JUnit output provides a terse summary of the number of tests started, aborted, succeeded, and failed. That being said, JUnit provides a very good basis on which to build upon. In fact, JUnit is commonly used by computer science instructors and is a student learning objective in some curriculums.

Prior to the development of the mutools framework, the author worked around JUnit's limitations in several ways. One approach involved writing customized scripts to process the JUnit output and perform scoring related tasks. A second technique included adding extra code into the tests to compute scores and, for example, determine if a test completed. Both of these approaches are clumsy and error prone. These workarounds, however, exhibited common coding structures that were ultimately incorporated into the current framework. For example, operations performed at the beginning and ending of each test are now performed by the framework. This is possible because JUnit provides callbacks that are automatically invoked by JUnit, which is utilized by mutools.

This lightweight library built on top of JUnit allows for fast development of JUnit tests that provide scoring and reporting. The approach requires the instructor to insert Java annotations into the source code that control scoring, partial credit, and reporting. Here are two example annotations that will be discussed in the next section. At this point, it suffices to say that the annotations are declarative statements that appear above the tests and the class definition.

```
@TestCategory(name="remove", points=10)
@GradingTest(category="removeByIndex", percentage=0.7)
```

IV. The Framework

A. Example

This section provides a complete example of a test suite that uses the mutools framework. The example is based on an assignment that might appear in a data structures course: implement a `List` data structure. Similar to AutoCode [8], a Java `interface` would be distributed to the students. Figure 1 shows the List interface. In this example, and subsequent code snippets, line numbers have been placed along the left hand side. Gaps between consecutive numbers will appear if one or more lines of source code were omitted. For example, line 001 is immediately followed by line 006. The missing four lines contain the documentation for the size method. These were omitted for the sake of conciseness. The github repository contains the entire contents of all files.

The `List` interface includes documentation that specifies the behavior of each of the data structure's methods. The documentation includes: what value is returned (if any), what exceptions should be thrown and under what conditions (if any), and how the data structure should be modified (if at all). Lines 020-026, for example, specify that the `get` method should return the list's value at the given index. Also, the method should throw an `IndexOutOfBoundsException` if the index is negative or exceeds the end of the list. Students use this documented interface to guide their implementation efforts and instructors use it to create tests that verify the implementation.

One approach to scoring requires the instructor to assign a point value to each method [2], where these values are communicated to the students in a separate document or within the interface itself. The size of the point values are usually based on a combination of: estimated difficulty of the method, whether code has been provided previously, etc. [12] Figure 2 shows a sample test suite that uses the framework. Again, only selected lines are shown. Lines 001-002 import the basic JUnit classes. Lines 013-018 define the categorical units of testing. Categories can be used to refer to a defined feature of the software. In this example, the `addAtIndex` category (line 018) refers to the collection of tests that verify the following method: `void add(int index, T item)`. Each category can contain multiple tests, where each individual test examines a particular situation that the instructor considers interesting. For example, one test can examine the behavior when adding to the interior of a lengthy list. A second test can verify adding to the end of a list and yet another test can be defined to add at the beginning of the list.

Lines 108-118 test the get method when the passed index is valid and the list contains one or more items. Line 108 specifies that this test is a member of the get category and is worth 70% of the category's points, 7 points in this case (i.e., 0.7*10). The `gradingScheme` is set to be `allornothing`, which is a predefined constant that indicates that all of the asserts performed must be processed correctly in order for any points to be awarded. The timeout attribute is a standard JUnit attribute and is defined to be 500ms in this example. With this attributed, a test is terminated if the student code enters an infinite loop or is inefficient. Within the test, a loop creates lists of size 1, 2, 3, 4, and 5 and verifies that each element in the list is properly

```
001 public interface List<T> {
006   public int size();
012   public boolean isEmpty();
018   public void clear();
020   /**
021    * Returns the element to the specified position.
022    * @param index position of targeted item
023    * @throws IndexOutOfBoundsException - if the index is
024    * out of range (index < 0 || index >= size())
025    */
026   public T get(int index);
036   public T set(int index, T item);
043   public void add(T item);
054   public void add(int index, T item);
055   /**
056    * Removes the first occurrence of item from this list if
057    * it is present.
058    * @param item The item to be removed.
059    * @return true if this list contained the specified element
060    */
061   public boolean remove(T item);
070   public T remove(int index);
078   public boolean contains(T item);
080   /**
081    * Returns first index of the specified element.
082    * @param item element whose presence in this list is to be
083    * tested.
084    * @return Returns first index of the specified element. -1
085    * is returned if item is not in list.
086    */
087   public int indexOf(T item);
097   public int lastIndexOf(T item);
104   public Object[] toArray();
105 }
```

Fig 1. List interface

retrieved by `get` (line 114) and verifies that the list has not been changed; line 115 invokes a helper method not shown in the code listing. Aside from line 108, the test resembles a regular JUnit test. The assert provided by mutools mirror those provided by JUnit. These include: assertTrue, assertFalse, assertEquals, assertNotEquals, assertNull, assertNotNull, assertSame, assertArrayEquals, assertThrows, and fail. The implementation of these revised asserts allow a test to continue when an assert fails. Additionally, they collect statistics regarding the number of asserts performed and the percentage of asserts correctly processed. Statistics are stored and later incorporated into a report.

Lines 119-138 define a second test for `get`. This test examines the exceptional conditions that a correct implementation must handle. In particular, negative indices and indices that exceed the end of the list are situations that should be considered. It is not uncommon for students to handle some, but not all, of the exceptional situations. In situations such as this, awarding partial credit may be sensible. [2,13] Line 119 defines a `gradingScheme`, which defines how the points are to be awarded. In this case, partial credit is to be awarded, where the amount of credit to award is defined using an interval notation. In this example, working right to left, if the student code correctly processes 100% of the asserts, 100% of the points are awarded. If the percentage of correct asserts is 50% or better, 60% of the points are awarded. The remaining intervals are similarly defined. It is worth noting that performing a similar task with standard JUnit is not readily done.

B. Reports

To illustrate the reports, a fictional student submission was used. This source code is not shown but is available on github. Several errors were purposely introduced: 1) `lastIndexOf` throws an exception when the called on the empty list and crashes if the key is the last element in the list. 2) `remove` enters an infinite loop when the item is not in the list. 3) `get` does not throw the proper exception when the index is equal to the length of the list.

At the completion of the test suite, a report is generated at several levels of detail: low, medium, and/or high. Line 013 defines which reports are to be produced. Figure 3 shows an

```
001 import org.junit.*;
002 import org.junit.runner.RunWith;
013 @Reporting(lowDetails = true, medDetails = true, highDetails = true)
014 @TestCategory(name = "clear", points = 3.0)
015 @TestCategory(name = "get", points = 10.0)
016 @TestCategory(name = "set", points = 10.0)
017 @TestCategory(name = "add", points = 10.0)
018 @TestCategory(name = "addAtIndex", points = 10.0)
027 @RunWith(GradingRunner.class)
028 public class LinkedListMUTests {
108    @GradingTest(category = "get", percentage = 0.7,gradingScheme="allornothing")
109    @Test(timeout = 500)
110    public void getListsLength1to5() {
111      for (int sz = 1; sz <= 5; sz++) {
112         for (int i = 0; i < sz-1; i++) {
113               LinkedList274<Integer> L = createList0toN_1(sz);
114               assertEquals((Integer)i, L.get(i));
115               assertTrue(listContains0toN_1(L, sz + 1));
116         }
117      }
118    }
119    @GradingTest(category = "get", percentage = 0.3,
120      gradingScheme="[0.0-0.3)=0.0,[0.3-0.5)=0.3,[0.5-1.0)=0.6,[1.0-1.0]=1.0")
121    @Test(timeout = 500)
122    public void getExceptions() {
123      LinkedList274<Integer> L = new LinkedList274<Integer>();
124      assertThrows(IndexOutOfBoundsException.class, () -> { L.get(-1); });
125      assertThrows(IndexOutOfBoundsException.class, () -> { L.get(0); });
138    }
300    @GradingTest(category = "indexOf", percentage = 0.35, gradingScheme = "code")
301    @Note(description="Creates lists of length 1, 2, and 5.\n" +
302                      "Cerforms multiple searches on each: " +
303                      "first element, second element, ... last element")
304    @Test(timeout = 500)
305    public void indexOfNoDuplicatesAndSuccessSearch() {
306      LinkedList274<Integer> L1 = createList0toN_1(1);
307      LinkedList274<Integer> L2 = createList0toN_1(2);
308      LinkedList274<Integer> L5 = createList0toN_1(5);
309
310      assertEquals(0, L1.indexOf(0));
311      assertTrue(listContains0toN_1(L1, 1));
312
313      assertEquals(0, L2.indexOf(0));
314      assertEquals(1, L2.indexOf(1));
315      assertTrue(listContains0toN_1(L2, 5));
323    }
431 }
```

Fig 2. List test suite.

```
Pts           Category                                Subtest  Corr      Asserts   Finish

=============================================================================================
8.8/10.0      get                                               no        61/77   normal
0.0/8.0       lastIndexOf                                       no        11/18 abnormal
8.0/10.0      remove                                            no          6/6  timeout
10.0/10.0     set                                              yes        36/36   normal
...
88.8/100.0    Totals                                         10/13      287/310
```

Fig 3. Excerpt from report at low detail.

```
Pts           Category                                Subtest  Corr      Asserts   Finish

=============================================================================================
8.8/10.0      get                                               no        61/77   normal
     1.8/3.0                                 getExceptions    no        31/47   normal
     7.0/7.0                          getListsLength1to5   yes        30/30   normal
---------------------------------------------------------------------------------------------
0.0/8.0       lastIndexOf                                       no        11/18 abnormal
     0.0/2.8  lastIndexOfWithDuplicatesAndSuccessSearch    no          3/6   normal
     0.0/2.8    lastIndexOfNoDuplicatesAndSuccessSearch    no         8/12   normal
     0.0/2.4               lastIndexOfUnsuccessfulSearch    no          0/0 abnormal
---------------------------------------------------------------------------------------------
8.0/10.0      remove                                            no          6/6  timeout
     8.0/8.0             withDuplicatesAndItemIsPresent   yes          6/6   normal
     0.0/2.0                removeWhereItemIsNotPresent    no          0/0  timeout
---------------------------------------------------------------------------------------------
...
88.8/100.0    Totals                                       10/13      287/310
```

Fig 4. Excerpt from report at medium detail.

excerpt of the report at the low level of detail. At this level, only a single-line summary of each testing category is displayed. The column labeled `Corr` indicates if all of the tests with that category were passed. The next column provides the number of asserts passed versus the number performed. The last column indicates if there was a problem in the student code terminating. In the given example, the student's remove method timed out at least once. And, `lastIndexOf` threw an unexpected exception.

Figure 4 shows an excerpt using the medium level of detail. The information includes all of low detail information but includes information regarding the individual tests that make up each category. In the given example, `lastIndexOf` was tested under three broad conditions: 1) when the list contains the key and the key is unique 2) when the list contains the key and there are multiple copies of the key 3) when the key is not present in the list. In this example, the tests examining conditions 1 and 2 completed without exception but processed only 3 of 6 assertions and 8 of 12 assertions, respectively. The test of `get`'s exception handling yielded 31 of 47 correct assertions for a 66.0% success rate. Referring back to line 120 from Figure 2, a 66% success rate is awarded partial credit of 60% of the test's points.

Figure 5 shows an excerpt using the high level of detail. This output provides detailed information about which particular asserts were passed/failed. This is useful only if a copy of the test suite is distributed to the students. Without that, the line numbers are useless. Within the `get` category, `getListsLength1to5` is fully passed. Assert statements on lines 114 and 115 are each executed 15 times. Referring back to lines 114 and 115 in Figure 2, it can be seen that these lines are in a loop and these asserts are performed a total of 15 times. The test named `getExceptions` verifies `get`'s handling of exceptional conditions. The detailed report shows that line 124 was correctly processed but line 125 was not. Referring back to lines 124 and 125 in Figure 2 shows that an `IndexOutOfBoundsException` was thrown with an empty list and a negative index but the student code did not throw an `IndexOutOfBoundsException` under the same conditions with the index of 0; in this case, the fictional student's code crashed with a `NullPointerException`.

```
Pts           Category                              Subtest  Corr      Asserts   Finish
=============================================================================================
8.8/10.0      get                                              no        61/77   normal
     1.8/3.0                                 getExceptions    no        31/47   normal
                                                        124                1/1
                                                        125                0/1
                                                        130               15/15
                                                        133                0/15
                                                        135               15/15
     7.0/7.0                           getListsLength1to5   yes       30/30   normal
                                                        114               15/15
                                                        115               15/15
---------------------------------------------------------------------------------------------
8.0/10.0      remove                                           no          6/6  timeout
     8.0/8.0             withDuplicatesAndItemIsPresent   yes          6/6   normal
                                                        237                1/1
                                                        238                1/1
                                                        239                1/1
                                                        240                1/1
                                                        241                1/1
                                                        242                1/1
     0.0/2.0                removeWhereItemIsNotPresent    no          0/0  timeout
---------------------------------------------------------------------------------------------
...
88.8/100.0    Totals                                    10/13      287/310
```

Fig 5. Excerpt from report at high detail.

V. Experience Using the Framework

A. *Efficacy*

The main contribution of this work is the development of a software framework that allows JUnit tests to be adapted quickly for grading purposes. When an instructor deems that unit testing is an appropriate approach for the given assignment, the framework can provide a rapid way to convert existing tests into ones that support common grading tasks. In these situations, the instructor develops JUnit tests as usual. This development cost is incurred whether or not the mutools framework is used. Once developed, some basic steps need to be performed: altering several import statements at the top of the tester, adding the desired @TestCategory annotations to the testing class's header, and then adding @GradingTest annotations to each of the individual tests.

For the sample test suite presented in the previous section, these additions consisted of approximately 40 lines of code being modified/added. Performing these modifications/additions required approximately 15 minutes of the author's time. Was the extra effort worth doing? The answer is yes if scoring, partial credit, and/or reporting are important to the instructor. Accomplishing these tasks in any other manner is difficult and error prone. Past experience indicates approximately four times as much time is needed to provide the same facilities using standard JUnit (e.g., scripts to post process JUnit output or adding custom code to each test).

From the student's point of view, the reports produced are conventional. At the lowest level of detail, students see which top-level features of their code were problematic. This provides students information regarding the methods that have errors. The scores associated with them also provide information about the severity of the errors. At the medium level of detail, students see the individual tests that succeeded and failed. The names of the tests and the descriptions of the tests provide students additional information about the conditions that caused errors and likely places in their code that need attention. In practice, the medium level of detail is preferred by the students. The high-detail report provides more detailed information about how the errors were exposed in their code. In particular, the line numbers can be used by the students to index into the test suite and see what calls led to their errors. The high level detail only if the instructor provides the testing code to the students; otherwise, the line numbers provided are meaningless. In practice, very few students run the tests themselves and fix their code. This observation would undoubtedly change if a follow-on assignment required them to fix their errors.

B. *Other Considerations*

Experience using the framework led to small changes in test development. In particular, it has been found that testing a particular feature should be divided into more individual tests, where each individual test examines a particular situation that might cause student code trouble. For example, some interesting conditions of `lastIndexOf` are: 1) when the key is not present in the list 2) when the key is present and is unique 3) when the key is present and there are multiple copies of the key. By dividing tests into small units, a more accurate assessment can be made in the event that the student code crashes. For example, if the student code crashes in scenario 3, the tests for cases 1 and 2 will still have at least completed. In contrast, if all

of the testing were put into one large test, the results would show the one test with abnormal termination. And, depending on when the exception was thrown, the other working parts of their code might not be exercised at all. Thus, some care should be taken when developing the tests.

The test suite presented in the previous section is strictly a functional test. It did not examine the details of the underlying implementation. For example, if the assignment required the use of a linked list, the tests would pass even with an array-based representation. In these situations, the test suite can include helper methods to verify the integrity of the underlying data structure. For example, the helper method `listContains0toN` could traverse the linked list starting with `head` and determine if the list contains the desired elements. Or, if a circular doubly linked deque were used, the circular nature of the deque could be verified by inspecting the `next` and `prev` fields. Developing tests to do this requires students to use consistent names for their data members, in addition to exposing the data members to clients. Providing such a structure places restrictions on the design, which can be problematic since the students have less design choices [14]. Aside from checking if linked data was used, [13] considers object oriented programming issues such as the use of inheritance.

The process of testing is a significant focus in many introductory computer science courses. After students are introduced to unit testing in early assignments, they can be encouraged to develop their own unit tests concurrently with their implementation. This test driven development (TDD) [15] process is to: create tests for a new feature and implement that feature so that the tests are passed. The goal is to improve software quality. As students gain experience, their tests become more sophisticated and TDD becomes a realistic expectation. Prior to attaining this proficiency, students express that they want the final test suite to be released when the assignment is distributed. This, of course, allows them to directly code to the tests, which may not be conducive for students' development. Eventually students need to develop the tests on their own.

Contract grading or *specifications grading* is becoming more common in computer science education because they provide an alternative way to grade that is thought to be more equitable. [16] reviewed the relevant literature to distill the aspects relevant to these approaches. Allowing resubmission is one of the approaches. The current framework can obviously assist with this approach because the grading process is more efficient. Within this category, there are issues related to: the number of times resubmissions are allowed, when they are accepted, what penalty to apply, etc. Binary grading (i.e., mastery or non-mastery) or bucket grading (i.e., classifying the submission into a well-defined category) are not directly supported by the current framework but could be in the future.

The development of a robust and thorough test suite is a critical, but difficult, part of the testing process. The instructor's expertise is the most important contributor to the quality of the test suite; however, additional techniques can be used to improve the quality of the tests. In particular, mutation testing can be used to improve the test suite [17]. In this approach, small systematic modifications are made to the component undergoing testing. These modifications introduce errors that should be uncovered by the tests. If not, the tests are improved to the point where the mutants are revealed. The use of mutation testing can be performed with any form of unit testing, including the framework presented here.

VI. Conclusions and Future Work

The mutools framework has been used several semesters and has scored approximately a thousand submissions across many different assignments. This framework allows instructors to quickly adapt JUnit tests into an equivalent test suite that computes scores and produces testing reports that can be distributed to the students. The framework provides facilities to compute scores and award partial credit. Using only standard JUnit to do a similar task is much more time consuming and error prone. This framework is available on github [18].

Planned future work addresses the problem of student work being mostly correct except for one or two methods. Others have addressed the problem of fixing code [2,19,20]. In the current context, a somewhat common occurrence is when the bulk of the implementation relies on one or two methods. When those methods are incorrect, the other methods will fail even though their logic is correct. For example, inserting an item to the end of a list can be implemented using a call to: `insert(size(), item)`. Thus, if `insert(int index, T item)` is faulty, appending to a list will also be incorrect. Fixing their code requires a correct implementation of inserting at an index. In the event of errors, the framework reruns the tests with one or more student methods replaced by a correct version. The framework would then display the test results with the original and the fixed code. This extension is nearing completion.

References

[1] G. Hagerer, L. Lahesoo, M. Anschütz, S. Krusche and G. Groh, "An analysis of programming course evaluations before and after the introduction of an autograder," 19th International Conference on Information Technology Based Higher Education and Training (ITHET), Sydney, Australia, 2021, pp. 1-9.

[2] D. Insa and J. Silva, "Automatic assessment of Java code," Computer Languages, Systems & Structures, vol. 53, 2018, pp. 59–72.

[3] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," ACM Transactions on Computing Education, vol. 19, 2019, pp. 1–43.

[4] J. Paiva, J. Leal, and A. Figueira, "Automated assessment in computer science education: A state-of-the-art review," ACM Transactions on Computing Education, vol. 22(3), 2022, pp. 1–40.

[5] C. Iddon, N. Giacaman and V. Terragni, "GRADESTYLE: GitHub-integrated and automated assessment of Java code style," IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), Melbourne, Australia, 2023, pp. 192-197.

[6] C. Tudose, JUnit in Action, Third Edition. Manning. 2020.

[7] O. Adeyemi, A. Adiraju, S. Akins, K. Khademi and B. Hui, "JUnit++: an open educational tool for simplifying unit testing," IEEE International Conference on Advanced Learning Technologies (ICALT), Orem, UT, USA, 2023, pp. 24-25.

[8] M. T. Helmick. "Interface-based programming assignments and automatic grading of Java programs," SIGCSE Bull. vol. 39(3) 2007, pp. 63–67.

[9] F. Jurado, M. Redondo, and M. Ortega, "Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice," Journal of Network and Computer Applications. vol. 35, 2012, pp. 695–712.

[10] J. Gao, B. Pang, and S. S. Lumetta, “Automated feedback framework for introductory programming courses,” Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2016, pp. 53-58.

[11] X. Liu, S. Wang, P. Wang, and D. Wu, “Automatic grading of programming assignments: an approach based on formal semantics,” IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), Montreal, QC, Canada, 2019, pp. 126-137.

[12] C. Douce, D. Livingstone, and J. Orwell, “Automatic test-based assessment of programming: A review,” Journal on Educational Resources in Computing, vol. 5(3), 2005, pp. 1–13.

[13] D. Insa and J. Silva, “Semi-automatic assessment of unrestrained Java code: A library, a DSL, and a workbench to assess exams and exercises,” Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2015, pp. 39–44.

[14] R. Acuña, T. Baron and S. Bansal, “Autograder impact on software design outcomes,” IEEE Frontiers in Education Conference (FIE), College Station, TX, USA, 2023, pp. 1-9.

[15] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep and H. Erdogmus, "What do we know about test-driven development?," IEEE Software, vol. 27(6), 2010, pp. 16-19.

[16] B. Harrington, A. Galal, R. Nalluri, F. Nasiha, and A. Vadarevu, “Specifications and contract grading in computer science education,” In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE). Association for Computing Machinery, New York, NY, USA, 2024, pp. 477–483.

[17] M. Betka and S. Wagner, "Extreme mutation testing in practice: An industrial case study," IEEE/ACM International Conference on Automation of Software Test (AST), Madrid, Spain, 2021, pp. 113-116.

[18] https://github.com/zmudam34/mutools

[19] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, 2017. “Automatic grading and feedback using program repair for introductory programming courses,” Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE). Association for Computing Machinery, New York, NY, USA, 2017, pp. 92–97.

[20] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013, pp. 15-26.