2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC
Host Sponsor UNC Charlotte
ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION
Paper ID #51413

A Software Framework for Testing Java Software Using Benchmark Code

Michael A Zmuda, Miami University

Michael Zmuda is an Associate Professor in Miami University's Department of Computer Science and Software Engineering. He earned a Ph.D. in Computer Science and Computer Engineering from Wright State University. His recent research focuses on developing software tools to assist computer science instructors.

A Software Framework for Testing Java Software Using Benchmark Code

Abstract

Testing student Java submissions is an important task performed by computer science faculty. JUnit testing is appropriate in many situations but can be cumbersome when the correct answer is not readily known or is too large to conveniently specify in the code directly. This paper presents one aspect of a software framework that assists instructors in these types of situations. In particular, this feature provides instructors the ability to automatically compare the output produced by the student's code against the output produced by an instructor-created solution, referred to as benchmark code. Within the instructor's solution, test cases are specified as a series of Java annotations appearing above the method. During testing, the student's method is executed on all test cases and its output is compared to the output produced by the benchmark code. Proper exception handling is also examined as well as verifying that input parameters are modified correctly or left unchanged. The framework includes report generation and scoring to lessen the burden of testing. The framework is available as a jar file on github.

Introduction and Related Work

Teaching computer science courses requires many time-consuming tasks to be performed. Large section sizes can cause faculty workloads to be heavy. The workload also increases when the number of assignments is large. Ideally, the number and scope of the assignments should be based purely on pedagogical reasons, not based on the need to reduce the time needed to grade the assignments. While the use of teaching assistants (TAs) can help in this regard, TAs do not solve the problem entirely because the faculty member is still responsible for developing the grading software and rubrics. For these reasons, an important objective is to provide instructors with tools to assist in doing these tasks quicker and more effectively. Toward this end, scholarly venues provide outlets for this type of work. ACM Conference on Innovation and Technology in Computer Science Education welcomes papers describing educational tools developed for use in computing education [1]. The American Society of Engineering Education's annual conference solicits papers in all areas of engineering education, including the use of computational tools in education [2]. Frontiers in Education [3] publishes papers that describe innovative practice in the area of education.

Tools developed for computer science education directly assist students in learning how to do software development. Also, a large body of work focuses on providing assistance to the instructor, which is the focus of the current work. Tools focusing on the instructor side of grading are often referred to as Automated Assessment (AA) tools [4]. In addition to lightening the workload, automation can provide more consistent and correct feedback to the students and can improve interactions with TAs [5]. Within the AA domain, tools have been developed to analyze

student code structure [6] and assess programming style [7]. One of the most common AA categories is providing feedback on test failures, which is the focus of this paper.

Online programming environments such as Kattis [8-9] and CodingBat [10] provide students with a catalog of problems to solve. These sites then judge the student's work. Some positive aspects of these online systems are that they are readily available, provide immediate feedback, and allow essentially unlimited practice opportunities. Kattis and similar sites judge student submissions based on the output produced by the entire program. CodingBat, on the other hand, presents the user with incomplete methods that are to be filled in with Java or Python code. CodingBat provides a teacher mode that allows instructors to assign problems and receive their students' testing results. Sites of this type provide a textual description of the task and at least one sample that illustrates the intended behavior. Judging uses the provided data and sequestered data to judge the student's submission. Typical judgements are: accepted, compile error, incorrect answer, runtime error, or timeout. Unlike the work described here, these systems do not consider exception handling, changes to the input arguments, or partial credit.

Some AA tools integrate scoring support [6, 11]. The work in [6] uses Java introspection to analyze and possibly fix student code. The work by Rossi [11] is somewhat aligned with the current work. Assigning partial credit for imperfect code is one shared aspect, although there are differences in the details. The current work adds features not appearing in Rossi's work: support for exceptions, support for non-primitive data, input argument verification, and use of benchmark code to define correct behavior. Conversely, Rossi's work includes features not appearing in this paper such as: clustering of errors across all submissions and support for graphical programs.

JUnit [12] is a widely adopted framework used to identify the presence of errors in code. Although designed for professional software development projects, it has been adapted for use in educational settings [13-15]. In educational contexts, instructors use JUnit methods such as assertEquals, assertThrows, and assertTrue to compare the behavior of student code with what the code should have done. Generally, the testing software will include dozens to hundreds of asserts. This approach is satisfactory when the goal is to simply establish whether a defect is present or not, but cumbersome in educational situations [13]. For example, some testing situations involve large input structures and/or non-primitive data that is awkward to specify in code. Also, the correct answer might not be known at compile time.

This paper presents one aspect of a software framework, mutools, that is designed to assist computer science instructors with grading student's Java code. The framework has similarities to JUnit but provides an improved approach to specifying the tests. First, the correct output is defined by the instructor's solution, referred to as *benchmark code*. Second, test cases are specified using a syntax that is easier to write than when using JUnit alone. Lastly, the framework abstracts away many of the details that are part of the testing process. While similar

testing can be achieved using JUnit, the approach described here accomplishes the task using a more convenient and reliable syntax.

Framework Goals

The framework presented in this paper assists instructors in the process of developing testing code. Students do not directly interact with the framework. Instead, students develop and test their code as would normally do. So, students do not need to install new software or learn a new tool. Instead, the framework is used by the instructor during the grading process to help reduce time to produce the tests, while also improving the reliability of those tests.

Currently, the framework handles only static Java methods. Methods of this type are evaluated on the following criteria:

- Correct output - For method that returns a value, does the returned value match the expected value? The current framework allows both primitives (e.g., int or float) and complex values to be returned (e.g., a Map or Set) and/or passed as parameters.
- Timeouts - Methods should take a reasonable amount of time to process the input. If the method does not terminate in a specified time interval, it is stopped and given the timeout classification. The goal here is to detect if the student's code has an infinite loop or is, for example, using an O(N) algorithm when an O(log N) is possible. The allotted time is a configurable quantity to account for large inputs.
- Exceptions - Some methods require specific exceptions to be thrown in specific situations. Alternatively, some methods should never throw an exception. In both cases, the student code's behavior should match the specifications.
- Arguments - An often overlooked aspect of testing is ensuring that the input parameters are correctly processed (or left unchanged). For example, a method that sorts an input array to solve a task unrelated to sorting should be flagged as incorrect. In many cases, the input parameters should remain unchanged after processing. There are also situations where the problem requires the input parameter(s) to be altered.

Expressing the correct behavior in all four categories described above can be cumbersome to do using text embedded in the testing code. To simplify the process, the framework uses benchmark code written by the instructor. The benchmark code operates correctly in all of the test conditions that the student code is expected to handle. During testing, the framework compares the behavior of the student's code with the behavior of the benchmark code. All four of the categories listed above must be passed in order for the test to be passed.

To promote its reuse by other instructors, the framework is distributed as a Java jar file on github [16]. This lightweight framework does not need to be installed and a server is not required. It can be easily integrated into existing test scripts using commands to compile and run the code. The following can be used if the benchmark code resides in PlacesSolution.java:

```
javac -cp mutools.jar *.java
java -cp ".:mutools.jar" PlacesSolution > report.txt
```

Example Assignment Using the Framework

This section presents a small sample assignment, the tests, and benchmark code created by the author. Fictional student work, with errors purposely introduced, is used to illustrate the framework's ability to process both correct and incorrect methods. Additionally, the example shows the syntax used by the instructor to configure the tests. Lastly, the reports generated are shown. Since the full example contains approximately 450 lines of code, only the most important parts are presented here. The full source code files are located on github.

The framework and the following sample assignment provide a concrete solution of the ideas proposed in this paper. The main suggestion is that there is benefit to using different testing syntax when working within an educational setting. It is understood that alternative syntaxes could be designed and other features included. As such, the presented library is simply an elaboration of the proposed ideas and is not necessarily an ending point. That being said, the framework exists and is ready to use.

The assignment presented here is one that might appear when introducing Java's standard data structures. This assignment ensures that students gain experience creating Java's Set and Map objects. The application domain for this assignment is based on US zip codes and their associated cities and states. A csv data file contains over 81,000 locations defined by the US post office. Five pieces of information are included for each location: zip code, city name, state abbreviation, latitude and longitude. The students' objective is to write methods that process these locations and create Sets and Maps for various purposes. The entire example includes 5 source code files and one csv file. Here are brief descriptions of these files:

- zipcodes.csv - This file contains data corresponding to zip codes in the United States in csv format. There are five columns: zip code, city name, state abbreviation, latitude and longitude. The zip code is represented as an integer, city and states are strings, and latitude and longitude are floating point values.
- Location.java - This class represents a single line of the data file. This basic class definition contains five data members, one for each of the five columns in the csv file. Students will not change this file.
- PlacesUtilities.java - This file contains code to read the data from the csv file into an ArrayList of Locations. The file also includes useful methods to display large data structures without exceeding 80 columns. Students will not change this file.
- PlacesStudent.java - This is where students write their code. When initially distributed to the class, this will contain several documented static methods that the student must implement.

- PlacesSolution.java - This is developed by the instructor and is not distributed to the students. This file mirrors the student's source code but has correct implementations of the methods inserted. Additionally, test cases are added to each method.

The data in zipcodes.csv is shown below. The first line is a table header, which is simply discarded during the input process. The given input routine parses the table's rows and places the data into an ArrayList of Locations. It is worth noting that a single zip code can be known as multiple names. For example, the zip code 10474 is referred to as: Bronx or Boulevard.

```
Zipcode,City,State,Lat,Long
...
10474,BRONX,NY,40.84,-73.87
10474,BOULEVARD,NY,40.84,-73.87
10475,BRONX,NY,40.84,-73.87
10499,BRONX,NY,40.84,-73.87
10499,PRIORITY MAIL CTR,NY,40.84,-73.87
10001,NEW YORK,NY,40.71,-73.99
...
```

To illustrate the different capabilities of the framework, five methods are assigned to the students:

1. `public static Set<String> getCitiesByZipcode(int zipcode)`
   This method receives a zip code (e.g., 10474) and returns the set of names associated with that zip code (e.g., a set containing "Bronx" and "Boulevard)."
2. `public static Map<String, Set<String>> getCityNames()`
   This method returns a map keyed by state abbreviation, with the keys' values set to the Set of city names in that state.
3. `public static Set<String> commonCityNames(String state1, String state2)`
   Returns the set of city names that appear in both of the given states.
4. `public static Set<String> statesWithThisCity(String city)`
   Returns the set of states that contain a specific city name. The problem specification requires an IllegalArgumentException to be thrown if city is null.
5. `public static boolean statesAreLegalAndUnique(String [] states)`
   Determines if an array of states contains a collection of unique, legal, state abbreviations.

In the fictional student's work for this example, methods 1 and 2 are correct, while methods 3-5 have errors introduced. The following summarizes the fictional student's implementation and main purpose of the example:

- `getCitiesByZipcode` - Correctly implemented.
- `getCityNames` - Correctly implemented.
- `commonCityNames` - The student's method is mostly correct but produces incorrect output when AK (i.e., Alaska) is either of the inputs and an infinite loop is entered if TX (i.e., Texas) is an input.
- `statesWithThisCity` - This method requires specific exceptions to be thrown in certain cases. The student's code does process exceptions correctly in some cases.
- `statesAreLegalAndUnique` - The student's method returns the correct value but changes the input array (i.e., sorts the array) when the method description states that the input argument should be left unchanged.

Figures 1a and 1b illustrate the structure of the student's work and the instructor's solution. The instructor defines and documents several methods that the students must implement. This includes JavaDoc to describe the method's intended behavior and the method's signature (name, return type, and formal arguments). The instructor's solution mirrors the students' work. All of the methods are solved by the instructor and serve as the benchmark code. Test cases precede each method and are specified using customized Java annotations. Lastly, the solution's main method launches the testing procedure by specifying the student's class followed by the instructor's testing class. Once launched, the framework executes both student and instructor code for each method, using all provided test cases. A report is generated and distributed to the student. The following paragraphs examine each of the student's five methods and the reporting produced by the framework.

```
public class PlacesStudent {
     /* JavaDoc for getCitiesByZipcode*/
     public static Set<String> getCitiesByZipcode(int zipcode) {
          // student's code goes here
     }
     /* JavaDoc for getCityNames */
     public static Map<String, Set<String>> getCityNames() {
          // student's code goes here
     }
     …
     public static void main(String[] args) {
          // student-written code to test all methods.
     }
}
```

Figure 1a. Structure of student code - PlacesStudent.java.

```
public class PlacesSolution {
     /* test cases for method1 appear here */
     public static Set<String> getCitiesByZipcode(int zipcode) {
          // solution code
     }
     /* test cases for method1 appear here */
     public static Map<String, Set<String>> getCityNames() {
          // solution code
     }
     …
     public static void main(String[] args) {
          mu.testing.Runner.testByCode(PlacesStudent.class,
                                       PlacesSolution.class);
     }
}
```

Figure 1b. Structure of solution code - PlacesSolution.java. The solution mirrors the student's code structure but includes the test cases to be conducted. The tester is launched in the main by specifying the class names of both the student's class and the testing class.

Figure 1. Structure of student and solution code.

Figure 2 shows the details for the first method, getCitiesByZipcode. Figure 2a shows the student's work, which is correct. It creates a HashSet, iterates over the list of Locations, and adds those cities that have a zip code matching the target zip code. Figure 2b shows that the instructor's code is similar but uses a TreeSet to provide sorted output. Regardless of this difference, the Sets produced by the two methods are compared for logical equality. And, in this case, the student's method is deemed to be correct. The instructor's solution uses framework-defined Java annotations to control the testing process. The annotation @Problem identifies a method that should be tested in addition to allocating points to this method, which is 20 points in this case. A second annotation type, @TestCase, defines a test to be conducted. This notation is somewhat similar to the use of tags appearing in JavaDoc sections used in [11]. For the first @TestCase, the parameter io="(10005)" indicates that the method is to be called with a one integer parameter representing the zip code 10005. Two additional tests are also included. It should be noted that the io definition must match the signature of the associated method. For

example, the method f(int a, double b) would require a TestCase involving two parameters: @TestCase(io="(1,3.14)"). The framework uses Java's reflection capabilities to match the actual parameters provided with the data types expected by the method signature. Isolating the test case syntax in this way provides a clean interface that reduces the likelihood of errors. Additionally, the TestCase syntax supports inputs as arrays, Sets, and Maps.

Figure 2c shows the report produced when testing this method. The method name is displayed along with the number of points awarded. In this case, all 20 points are awarded because the student's code correctly processed all three test cases. The next three lines show the test data and a message showing the student code operated correctly. The last line of this section includes a summary line.

```
public static Set<String> getCitiesByZipcode(int zipcode) {
     Set<String> result = new HashSet<>();

     for (int i=0; i<places.size(); i++) {
          if (places.get(i).zipCode == zipcode) {
               result.add(places.get(i).cityName);
          }
     }
     return result;
}
```

Figure 2a. Student implementation for getCitiesByZipCode. This implementation is correct.

```
@Problem(pts=20)
@TestCase(io="(10005)")
@TestCase(io="(90002)")
@TestCase(io="(999999)")
public static Set<String> getCitiesByZipcode(int zipcode) {
     Set<String> result = new TreeSet<>();
     for (Location loc : places) {
          if (loc.zipCode == zipcode) {
               result.add(loc.cityName);
          }
     }
     return result;
}
```

Figure 2b. The test cases and benchmark code for getCitiesByZipCode. There are 3 test cases in this example.

```
getCitiesByZipcode 20.0 points (out of 20)
io=(10005) Correct
io=(90002) Correct
io=(999999) Correct
Correct: 3   Incorrect: 0
```

Figure 2c. The output produced by the framework when testing this method. The framework iterates over the given test cases and compares the student's output against the solution's output. For this method, all three test cases produce the correct answer.

Figure 2. getCitiesByZipcode

The second method - getCityNames - is also correctly implemented by the student. For conciseness, the code is not shown. It is worth noting that the correct output is a large Map that would have been very difficult to specify without using benchmark code. The testing report for this method appears as:

```
getCityNames 20.0 points (out of 20)
io=() Correct
Correct: 1   Incorrect: 0
```

The student's third method - commonCityNames - includes several flaws. First, whenever Alaska is involved, null is returned. Second, when Texas is involved, the algorithm takes an excessively long time to terminate (e.g., infinite loop). The testing for this method allows for partial credit by specifying a grading scale to be used on this problem. This partial credit feature has similarities to what is proposed in [11]. In PlacesSolution.java, the @Problem annotation looks like:

```
@Problem(pts=20,
gradingScheme="[0-0.75)=0.0,[0.75-1.0)=0.7,[1.0-1.0]=1.0",
runningTimeMS=500)
```

This grading scheme defines three intervals: [0, 0.75), [0.75, 1.0) and [1.0, 1.0]. The endpoints of each interval represent the percentage of test cases passed. For this problem, the instructor specified 10 test cases, so this translates to the following three intervals: [0, 7.5), [7.5, 1.0), and [1.0,1.0]. This loosely represents: a method with many errors, a method with a few errors, and a correctly written method. Each interval also defines a percentage of points to be awarded: 0%, 70%, 100% (i.e., 0, 14, or 20). By default, each method must complete within 100ms; otherwise, a timeout is recorded and the method is terminated. For this method, 500ms is allocated. The testing report is shown below, where the test cases involving Alaska and Texas are each flagged with an informative message.

```
commonCityNames 14.0 points (out of 20)
io=(`OH`, `MI`) Correct
io=(`MO`, `IL`) Correct
io=(`MA`, `MI`) Correct
io=(`AK`, `OH`) Incorrect
 returned: null
should be:
[`ANDERSON`,`BEAVER`,`BETHEL`,`BUCKLAND`,`GALENA`,`GUSTAVUS`,`HO
MER`,`HOUSTON`,`PETERSBURG`,`SAINT MARYS`,`ST MARYS`,`STERLING`]
io=(`CA`, `MI`) Correct
io=(`TX`, `ME`) timed out
io=(`CT`, `FL`) Correct
io=(`DE`, `CT`) Correct
io=(`badstate`, `MI`) Correct
io=(`NY`, `badstate`) Correct
Correct: 8   Incorrect: 2
```

The fourth method - statesWithThisCity - uses a gradingScheme named "code," which is a symbolic name for the following intervals:

```
gradingScheme="[0-0.75)=0.0,[0.75-1.0)=0.80,[1.0-1.0]=1.0"
```

When a gradingScheme is not specified, all of the points are awarded only if all of the test cases are correctly processed; otherwise, 0 points are awarded. The student's code for this method includes an error due to a `NullPointerException`, when an IllegalArgumentException should have been thrown. The testing report for this method appears as:

```
statesWithThisCity 16.0 points (out of 20)
io=(`Oxford`) Correct
io=(`Springfield`) Correct
io=(`Detroit`) Correct
io=(`somecityname`) Correct
io=(null) Incorrect exception:
java.lang.NullPointerException----statesWithThisCity line 36
Correct: 4   Incorrect: 1
```

The fifth method - statesAreLegalAndUnique - accepts a Java array containing state abbreviations. The problem specification states that the incoming array should remain unchanged. In this case, however, the student code sorts the array during processing. Aside from that flaw, the student code is correct. During testing, the framework makes deep copies of all input parameters. After calling both the student method and the benchmark method, several comparisons are made. First, the output values are compared from equality. Second, the resulting input parameters are also compared. Thirdly, in the case where exceptions are involved, the exceptions must match. In this case, the student's incoming array will not match the solution's array if the input array was originally unsorted. The report generated includes messages indicating that the first parameter was changed improperly:

```
statesAreLegalAndUnique 0.0 points (out of 20)
io=([]) Correct
io=([`OH`]) Correct
io=([`NY`]) Correct
io=([`MI`, `OH`]) Correct
io=([`OH`, `MI`]) Argument #1 was [`MI`,`OH`] (should be
[`OH`,`MI`])
io=([`NY`, `XYZ`]) Correct
io=([`NY`, `MI`, `FL`, `NY`] Argument #1 was
[`FL`,`MI`,`NY`,`NY`] (should be [`NY`,`MI`,`FL`,`NY`])
Correct: 5   Incorrect: 2
```

Experience Using the mutools

The presented framework includes other features beyond those described in this paper. Across all features, the author and colleagues have processed thousands of student submissions. When considering only the benchmark aspect, a few hundred submissions have been processed. This experience has shown that doing similar testing tasks using JUnit alone is more cumbersome and more error-prone. Here, basic JUnit is used as a point of comparison because it is most closely aligned with the current work.

When assessing tools used in teaching, it is common to measure student learning with and without the intervention. That assessment approach does not apply in this work since the students do not interact with the framework. Since the software is designed to provide time savings for the instructors, one can measure the time taken to create tests using JUnit alone against the time taken to create the equivalent set of tests using the framework. Creating a fair experiment is challenging because some testing situations offer very little advantage one way or the other. The author's experience shows that reasonably complex assignments, such as the one described above, is well-suited to the framework and especially cumbersome for JUnit. The following paragraphs provide evidence that using JUnit alone would be more difficult to develop than when using the approach proposed here.

The simplest tests require only return values to be compared. Of course, this could be easily done in JUnit by:

```
assertEquals(student.getCitiesByZipcode(10005),
                solution.getCitiesByZipcode(10005));
```

It is correct to say that the JUnit approach requires approximately the same effort as the proposed approach. However, one can argue that minor reliability issues are introduced by having the same parameter list appear twice. In contrast, using the @TestCase notation, the input parameter appears exactly once. Handling timeouts requires roughly the same effort in both testing approaches.

When considering the more complex situations, JUnit is generally more cumbersome. Consider exception handling. JUnit does provide facilities to compare thrown exceptions using assertThrows but the code requires multiple lines if several different types of exceptions and exceptional situations are involved. For the framework, the exceptions thrown are defined by the benchmark code and therefore are not required to be specified in the @TestCase definition. Of course, in any testing approach, test cases must be included to cause the exceptions to be thrown. The use of benchmark code is cleaner than using a series of assertThrows.

Checking to ensure that input parameters are processed correctly requires considerable overhead when using JUnit alone. For example, testing statesAreLegalAndUnique would require:

```
String [] states = new String [] {"NY", "MI", "FL", "NY"  };
String [] statesStudent = states.clone();
assertEquals(student.statesAreLegalAndUnique (statesStudent),
                solution.statesAreLegalAndUnique(states));
assertEquals(states, statesStudent);
```

The first assert checks the return values and the second assert checks to ensure that the input parameters have been processed correctly. This section of code requires more lines for each additional parameter. It is more convenient and reliable to succinctly specify the test as:

```
@TestCase(io="([`NY`, `MI`, `FL`, `NY`]")
```

Behind the scenes, the framework automatically makes a deep copy of the input array and compares all of the parameters after execution. The likelihood of introducing coding mistakes is greatly reduced, while also simplifying the instructor's work.

Scoring and partial credit are not performed by JUnit. To be sure, JUnit was not designed for testing within an educational context. Nonetheless, scoring is a task relevant to computer science education and the framework integrates scoring directly into the syntax.

In summary, some of the instructor's work is common to any testing approach used. For example, the instructor must write the solutions in any case. And, developing the test cases is also a common overhead. However, the manner in which test cases are specific makes a significant difference. When using JUnit alone, adding the necessary assertEquals, assertThrows, etc. takes considerably more time than writing just one @TestCase. Additional time is also required to add the assertEquals and make the deep copies needed to check the passed parameters. All of this requires additional time and also increases the likelihood of errors. In contrast, the @TestCase notation hides the details of all of the underlying checks. To perform scoring, partial credit, and report generation, the JUnit approach will take much more time since it is not designed to do this. Overall, developing grading software using the framework can be done using much less time and with more confidence than using JUnit alone. It is difficult to conceive of a scenario where it would be more convenient to use JUnit alone in an education setting.

Conclusions and Future Work

mutools is a lightweight framework for supporting grading of static Java functions. The use of this framework reduces the instructor's effort to create testing software for computer science courses. Additionally, the reliability of the resulting tests is improved. This framework does not require an installation process and can simply be compiled into the testing executable. It is available as a jar file on github [16].

The framework compares the results returned by methods written by students against a known correct implementation written by the instructor. The library's capabilities include: checking return values, considering timeouts, checking exceptions, verifying that input parameters are processed correctly, dealing with primitive and non-primitive inputs, performing scoring, and awarding partial credit. While some of these features are present in JUnit, mutools provides all of them into one library that is ready to use. The approach used to define test cases is clean, concise, and more easily done than JUnit. The syntax has been designed to abstract away the underlying comparisons that should be made and the deep copying of input parameters. Without this framework, all of the tests and setup must be manually performed by the instructor. This of course requires considerable time and is more likely to include errors.

The features described here support only static methods. Extending the framework to allow non-static methods would be useful when testing OOP implementations. The extended infrastructure would look different from what is described here. The proposed scenario would allow the instructor to compare a student implementation of a class against a correctly defined class. These standard tests would involve defining a sequence of method calls that would be invoked on both the student and benchmark code in lock-step. After each individual method invocation, the two objects' state would be compared in a manner similar to what is currently done. The main question is how to specify the sequence of method calls. With a clean syntax for specifying the method calls, one would expect to see significant improvements in test set development.

References

[1] Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 2. Association for Computing Machinery, New York, NY, USA, 2025.

[2] Proceedings of the American Society of Engineering Education (ASEE), Montreal, Canada, 2025.

[3] Proceedings of the IEEE Frontiers in Education Conference (FIE), Nashville, TN, USA, 2025.

[4] J. Paiva, J. Leal, and A. Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. ACM Trans. Comput. Educ. 22, 3, Article 34 (September 2022), 40 pages.

[5] 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," 2021 19th International Conference on Information Technology Based Higher Education and Training (ITHET), Sydney, Australia, 2021, pp. 1-9.

[6] D. Insa and J. Silva. Automatic assessment of Java code. Computer Languages, Systems & Structures, 53:59–72, September, 2018.

[7] C. Iddon, N. Giacaman and V. Terragni, "GRADESTYLE: GitHub-Integrated and Automated Assessment of Java Code Style," 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), Melbourne, Australia, 2023, pp. 192-197.

[8] E. Enström, G. Kreitz, F. Niemelä, P. Söderman and V. Kann, "Five years with kattis — Using an automated assessment system in teaching," 2011 Frontiers in Education Conference (FIE), Rapid City, SD, USA, 2011, pp. T3J-1-T3J-6.

[9] https://open.kattis.com/ (accessed March 9, 2026).

[10] https://codingbat.com/java (accessed March 9, 2026).

[11] T. R. Rossi, “Semi Automated Partial Credit Grading of Programming Assignments,” M.S. thesis, Dept. Computer Science., Univ. of New Hampshire., Durham, NH, 2016.

[12] C. Tudose. *JUnit in Action*, Third Edition. Shelter Island, NY, Manning Publications, 2020.

[13] O. Adeyemi, A. Adiraju, S. Akins, K. Khademi and B. Hui, "JUnit++: An Open Educational Tool for Simplifying Unit Testing," 2023 IEEE International Conference on Advanced Learning Technologies (ICALT), Orem, UT, USA, 2023, pp. 24-25.

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

[15] M. Zmuda, "A Software Framework Based on JUnit for Automated Software Testing in Computer Science Courses," 2024 IEEE Frontiers in Education Conference (FIE), Washington, DC, USA, 2024, pp. 1-8.

[16] https://github.com/zmudam34/asee2026 (accessed March 9, 2026).