A Case For Reflection In Autograding

Chad Hogg
chad.hogg@millersville.edu
Millersville University
Millersville, PA, USA

ABSTRACT

Autograders are programs written to analyze student work from formative assessments and produce both grades and constructive feedback for the benefit of instructors and/or students. Many strategies can be used to develop these programs. This paper demonstrates a strategy based on reflection that allows test cases to examine the internal state of objects and to create objects with arbitrary internal states. We report on our experiences using this technique in a CS2 course, where we found that students given autograders based on this strategy produce more correct solutions than those given autograders that rely on the public interfaces of student-written classes.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Social and professional topics** → *Student assessment*; *CS1*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Autograding, Feedback, Introductory Computing

ACM Reference Format:
Chad Hogg. 2024. A Case For Reflection In Autograding. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653534

1 INTRODUCTION

Students in early, programming-intensive computer science courses require frequent opportunities to practice the skills they are learning and timely, detailed feedback about the quality of their work. Creating good feedback is a very time-intensive task, as there are both many correct ways to solve a problem and infinitely many incorrect ways. Explaining why and how a program is broken requires both sufficient testing to discover all deficiencies and sufficient debugging to determine the root cause of each. As instructor workload increases it becomes infeasible to do this manually. To simplify this process, many faculty members rely on automated testing, which they either run themselves on student submissions while grading or make available to students through an autograding system so that they can follow a submit - improve - resubmit workflow.

ITiCSE 2024, July 8–10, 2024, Milan, Italy

ACM ISBN 979-8-4007-0600-4/24/07
https://doi.org/10.1145/3649217.3653534

Unit testing suites such as JUnit provide a quick and easy way to write test cases that are good at discovering deficiencies, at least for those deficiencies that are detectable from the tested code's public interface. When a professional developer encounters a failed test, they can read the source code of the test to determine the action that caused the test failure and the context in which that action was taken. Furthermore, they can attach a debugger to the test and step through it to determine exactly where and when anomalous behavior begins. This is not usually true for a student submitting their assignment to an autograding system, and it is too time consuming for an instructor grading tens or hundreds of assignments. Thus, in the grading environment these kinds of tests are not effective at quickly determining the root causes of deficiencies.

A student submitting their code to an autograding system may not have direct access to the test case or the necessary experience to effectively use it. They will only get meaningful feedback about test failures if the designer of the tests has built that feedback into them. Because novices make creative kinds of errors that experienced programmers would not have even considered within the realm of possible outcomes, it can be difficult to predict, detect, and provide helpful feedback about all potential failure modes.

It is advisable to design tests for students around the primary goal of isolating root causes and explaining the exact state of the system and action taken in each test. This requires that only one action be taken per test, which in turn requires a mechanism for setting up a desired context without using the student code directly. It also requires the ability to observe all state changes within an object to determine whether or not the action is achieving its post-conditions and maintaining its invariants.

Fortunately, student assignments in introductory courses are often fully specified and require students to solve problems in very specific ways. For these types of assignments it is not unreasonable to require that every student's class use the same fields, with the same names and types, and the same public methods, with the same names and signatures. This allows reflection to be used to set up explicit, isolated test cases for each use case of every method in a class. Reflection also provides other benefits, such as the ability to selectively compile and run tests against only the parts of a class that a student has written so far.

Using reflection to access private members of a class for testing purposes is not a novel idea, but nor is it in widespread use. The technique is not appropriate for professional developers writing production code, because in that environment tests should only verify the external, public interfaces of classes while ignoring their implementation details [4]. The contribution of this paper is to explain why this technique can be appropriate for testing introductory student assignments, to demonstrate how it might be done, and to present preliminary evidence that doing so improves outcomes.

2 AN EXAMPLE PROBLEM DOMAIN

We will be using an assignment from a CS2 course as an example. In it, students implement the `Sequence` abstract data type, as described by Michael Main [8]. This data type is an ordered collection of values with a single embedded pseudo-iterator called the *cursor* and the following operations:

`size()`	gets the number of elements
`hasCurrent()`	whether or not cursor is valid
`getCurrent()`	gets value of element at cursor
`start()`	moves cursor to beginning
`advance()`	moves cursor to next element
`addBefore()`	adds element before cursor
`addAfter()`	adds element after cursor
`removeCurrent()`	removes element at cursor

Most operations that depend on the cursor throw an exception if the cursor is invalid, which occurs when advancing past the last element and when there are no elements. But `addBefore()` and `addAfter()` instead add at the beginning or end of the collection, respectively, when the cursor is invalid. These methods also move the cursor to the new element in all cases. In addition to the basic ADT operations the class has a `toString` method that concatenates the elements, separated by spaces, and marks the current element (if one exists) by parenthesizing it.

We will primarily be discussing an implementation based on a singly-linked list. Fields include a reference to the `head` node, a reference to the `tail` node, a reference to the `cursor` node, a reference to the `precursor` node, and a count of the `size`. The first four fields all reference `Node<E>` objects, which contain a piece of data and a `next` reference. Students are provided the following class invariants, as well as preconditions and postconditions for each method:

1. The number of items in the sequence is maintained in `size`.
2. `head` refers to the first node, if any, or is **`null`**.
3. `tail` refers to the last node, if any, or is **`null`**.
4. If there is a current element, then `cursor` refers to its node and `precursor` refers to the previous node, if any, or is **`null`**.
5. If there is no current element, then `cursor` and `precursor` are both **`null`**.

The `precursor` field is necessary because a singly-linked list has no way to look backward from a node, and the `addBefore` and `removeCurrent` methods require making changes to the node before the `cursor` node. We require this exact design for the assignment because one of our learning outcomes is that students understand how to use a `cursor` and `precursor` pair to allow constant-time insertions before the current node in a singly-linked list. Because other methods will succeed even if `precursor` is invalid, and because it is an implementation detail with no way to view it externally, failure to maintain the correct value of the `precursor` is a very common student error, and one that is challenging to debug because the cause of the failure and where it becomes apparent are often separated by several steps. Maintenance of the `tail` reference has similar problems.

3 TRADITIONAL AUTOGRADER DESIGN

The simplest tests for an implementation of `Sequence` would construct a sequence, add and remove elements while moving the cursor around, and then compare the `toString` result to its expected correct value. If such a test succeeds we can have reasonably high confidence that the class is working correctly – at least from the perspective of its public interface. But if such a test fails we have very little information about the source of the failure. Someone who has access to the test could step through it in a debugger and manually inspect its state, but if it is run automatically for students the best we could do is explain the series of steps taken to them so that they could simulate the test on their own. It would also be difficult to provide any partial credit for the assignment, because the single test or small number of tests either entirely fail or entirely succeed.

A more appropriate, and more common, testing strategy would be to build a large library of tests, each of which exercises one particular scenario that the code needs to handle. For example, one test case might confirm that calling `addAfter()` on a sequence with several elements and no current element puts the new element at the end. Another test case would confirm that calling `addAfter()` on a sequence with several elements in which the last element is current also puts the new element at the end. Other test cases would cover calling `addAfter()` when the first element is current, when some middle element is current, and when there are no elements at all.

Each test case would construct a sequence, manipulate that sequence to get it into the state that allows a specific scenario to be tested, and then call the method that should be tested and confirm that the expected post-conditions have occurred. Figure 1 shows the skeleton of a test in this format that checks adding after when the sequence has several elements but no current. (The assertion failure messages are elided for space.) This test is insufficient in two ways:

First, it has a hidden dependency on other tests and can only pass if (a) adding after to an empty sequence, (b) adding after when the last element is current, and (c) advancing when the last element is current all work correctly. The `toString()` method must also work correctly. This means that it is impossible to give someone partial credit for a working implementation of this specific use case if any of its dependencies are failing. It also means that it is difficult to distinguish the root cause when this method fails, though adding assertions after each step could help.

Second, checking only the accessors of the class does not allow us to confirm that all class invariants are still true. For example, all of these assertions could pass with `tail` still referring to the node that contains 3, and there are no other assertions we could make that directly detect this. Nor are we able to confirm that `precursor` refers to the node containing 3, as it should.

In fact, the instructor's solution that was used for this assignment did not maintain all class invariants in all situations; when advancing past the end of the sequence it left the `precursor` at the `tail` rather than making it **`null`** as invariant 5 requires. (Because the `precursor` should never be used for anything when the `cursor` is null, an argument could be made that this invariant is unnecessarily rigid.) This deficiency was only discovered as part of this project for improving testing.

```
@Test
public void testAddAfterNoCurrent() {
  LinkedSequence<Integer> seq =
    new LinkedSequence<>();
  seq.addAfter(5); // "(5)"
  seq.addAfter(3); // "5 (3)"
  seq.advance();   // "5 3"

  seq.addAfter(2); // "5 3 (2)"

  assertEquals("5 3 (2)", seq.toString());
  assertTrue(seq.hasCurrent());
  assertEquals(2, seq.getCurrent());
  assertEquals(3, seq.size());
}
```

Figure 1: A sufficient but ineffective test.

4 REFLECTION

Both of the problems with the test of Figure 1 could be solved if we had direct access to the internal state of the object we are testing: we could set up an exact scenario without depending on methods that might be broken, and we could confirm that all class invariants hold and post-conditions of the method we are testing have been achieved.

Requiring students to explicitly expose their class's internal state by making implementation details public would be encouraging bad habits, and thus should be avoided. Fortunately, Java offers a library for reflection, with which we can reason about and directly modify the structure of code. Specifically, these libraries allow us to make private components of a class accessible and to get and set field values bypassing any accessor or mutator methods that might exist. See [7] for a thorough introduction to the concept.

Figure 2 shows how to give yourself access to a field, even if it is private, and then how to then read and write its value. In this, `seq` is an instance of `LinkedSequence<E>`, while `value` is an instance of `Node<E>`. Although it is not needed for the `LinkedSequence` example, the library also offers the ability to call private methods.

```
Field headField = LinkedSequence.class.
  getDeclaredField("head");
headField.setAccessible(true);

Node<E> head = (Node<E>)headField.get(seq);

headField.set(seq, value);
```

Figure 2: Basic reflection usage.

We find it most convenient to encapsulate all of this into a helper class with methods for reading and writing each field. To make the code for future examples fit more easily into a column, we will be referring to this class as simply `R`, short for `Reflector`.

5 USE OF REFLECTION IN AUTOGRADERS

Figure 3 shows a revised version of Figure 1 that uses reflection to test the same scenario. Lines 3 through 6 set up a linked list containing 5 and 3. Lines 7 through 13 create a `LinkedSequence` and set its fields to exactly what they should be when that sequence contains 5 and 3 and has no current element. Unlike the earlier version of the test, they do so in a way that does not depend on any part of the `LinkedSequence` class being implemented correctly.

Lines 15 through 16 call the method that we are testing – the only `LinkedSequence` method call that exists anywhere in this test. Lines 18 through 27 confirm that the sequence's `head` refers to the beginning of a linked list containing 5, 3, and 2. Lines 28 and 29 confirm that the first two nodes of the list are the same objects they had been. Lines 30 through 33 confirm that the other four fields contain exactly the values they should if the method's post-conditions were achieved and the class's invariants still hold. We can thus confirm that the method has not merely appeared to have worked but has definitively met its specification.

```
@Test
public void testAddAfterNoCurrent() {
  Node<Integer> second =
    new Node<>(3, null);
  Node<Integer> first =
      new Node<>(5, second);
  LinkedSequence<Integer> seq =
    new LinkedSequence<>();
  R.setHead(seq, first);
  R.setTail(seq, second);
  R.setCursor(seq, null);
  R.setPrecursor(seq, null);
  R.setSize(seq, 2);

  try { seq.addAfter(2); }
  catch(Exception exception) { fail(); }

  Node<E> node1 = R.getHead(seq);
  assertFalse(null == node1);
  Node<E> node2 = node1.getNext();
  assertFalse(null == node2);
  Node<E> node3 = node2.getNext();
  assertFalse(null == node3);
  assertTrue(null == node3.getNext());
  assertTrue(5 == node1.getValue());
  assertTrue(3 == node2.getValue());
  assertTrue(2 == node3.getValue());
  assertTrue(first == node1);
  assertTrue(second == node2);
  assertTrue(node3 == R.getTail(seq));
  assertTrue(node3 == R.getCursor(seq));
  assertTrue(node2 == R.getPrecursor(seq));
  assertTrue(3 == R.getSize(seq));
}
```

Figure 3: A more thorough, carefully-written test.

One feature of this test that may seem odd is the decision to avoid `assertEquals()` or other even more specific assertions. When writing tests for experienced developers who will have access to the source code for the tests themselves, the extra information from more specific assertions can be valuable for quickly diagnosing the cause of a failure. But when writing tests for inexperienced students who do not have access to the source code for the tests and may not even be aware of the technology on which the tests are built, this information would instead be confusing. Instead, it is incumbent on the developer of the test to write very detailed failure messages explaining exactly how the test was set up, exactly what action was being tested, and exactly what kind of deficiency was discovered. All the messages are elided from this figure because they are long and complicated. The one on line 19, for example, might look like "LinkedSequence's addAfter() method should not have set the head to null when called on a sequence with two elements, neither of which was current". Note also that the test is careful never to make any assumptions, such as the value of a field that should not have changed remaining the same or a value that should be non-null actually being non-null. It should thus be the case that every possible failure mode can be detected and result in an understandable message.

Writing the test of Figure 3 requires substantially more time and care than that of Figure 1, but if you are going to be helping dozens to hundreds of students debug their code, the clarity that it provides will save far more time than was invested.

Space constraints prevent us from demonstrating how, but the same reflection libraries allow us to write tests that confirm that required class components exist before we attempt to compile tests that depend on them and tests that confirm that classes have not been extended with additional fields. They also allow us to selectively compile and run tests for only those methods that a student has added to a class thusfar, which is irrelevant in this case because students are implementing an interface. These techniques also allow us to make much friendlier error messages in cases of severe disregard for specifications that would instead cause catastrophic failure for traditional autograders.

6 EVALUATION

To validate the effectiveness of this approach, we have analyzed data that our autograding system automatically collects each of the most recent three times that we have taught our CS2 course.

6.1 Description

Our CS2 course includes a series of ten "laboratory assignments", formative assessments in which students write interesting programs or components of programs to precise specifications. The primary goal of these assignments is for students to gain a deeper understanding of the language features or theoretical concepts relevant to the assignment through practice.

Each of these laboratory assignments comes with an autograder, to which students may submit successive versions of their work until they are satisfied with their grade or have reached the deadline. (Human review may adjust that grade upwards or downwards later.) Each time the autograder is run on a submission, it executes a series of tests and produces feedback for the student and instructor explaining what parts of the program did not satisfy the specifications.

Two of the assignments in this course involve implementing the `Sequence` ADT described in Section 2. In the first, the student's implementation must use a partially-filled, replaceable array to store the elements of the sequence, and two integers storing the size of the collection and the index of the current element, respectively. In the second, the student's implementation must use references to the head and tail of a singly-linked list, as well as references to the cursor and precursor nodes and the size.

Directly before embarking on each of these assignments we have worked together as a class to implement the List ADT using its respective data structure, so the primary challenges for students are to understand these data structures clearly enough to use them independently and to adapt their understanding to the differing requirements of the Sequence ADT. Students are provided with an exhaustively documented interface and the skeleton of a class consisting of fields, class invariants, and a working `toString()` method as described in Section 2.

All students are given the same autograder for the array-based implementation. This autograder uses tests like those described in Section 3. For each of its test cases it sets up a scenario using the student's code, executes the method being tested, and check whether or not the `toString` method returns the expected string. If they do not match, it prints a description of the steps it took, what it expected the string to look like, and what it actually did look like.

For the links-based implementation all of our students get precisely the same instructions and starter code, but half get an autograder based on the same approach as the array-based implementation (the "old" autograder) while the other half get an autograder that was developed using the techniques described in Section 5. The students were grouped by section, with the same instructor teaching both sections.

6.2 Results

Over the course of three semesters we had 47 students complete both the array-based assignment and the old version of the links-based assignment, and 45 students complete both the array-based assignment and the new version of the links-based assignment. A small number of other students were excluded from the study because they either did not make any submissions for one of the assignments or submitted the links-based assignment to both autograders, ignoring our instructions.

Our assignment submission and feedback system is AutoLab, developed at Carnegie Mellon University [10]. Each time that a student makes a submission it generates a document containing the student ID, submission number, submission date, and autograder output. From these documents we were able to extract the number of submissions, score of initial submission, and score of final submission for each student on both assignments. Means and standard deviations of these results are shown in Table 1.

We used LibreOffice Calc's implementation of a two-tailed Student's t-test with no assumption of equal variance to determine whether or not the differences between the old-autograder and new-autograder populations were statistically significant.

Table 1: Experimental Results: Number Of Submissions and Initial and Final Grades for Array-based and Links-based Assignments

	Array Implementation			Linked Implementation		
	# Subs	Ini Grd	Fin Grd	# Subs	Ini Grd	Fin Grd
Old Array + Old Linked Autograder						
Mean	17.6	31.3	88.0	6.6	31.0	81.1
StDev	17.2	29.9	25.3	5.8	35.3	25.0
Old Array + New Linked Autograder						
Mean	18.2	34.6	94.3	14.8	47.6	96.4
StDev	21.8	31.1	18.0	15.1	33.6	8.38

We see that students who would go on to receive the new autograder for the links-based implementation made more submissions to and earned higher grades on the array-based implementation than those students who would go on to receive the old autograder. These differences are, however, quite small, and not significantly significant. This is as expected, because there is no reason for us to believe that any true differences exist between these populations as regards the array-based implementation. The data is included here to confirm that the populations behave similarly when given identical autograders.

On the links-based implementation, however, there are noticeable differences. Students using the new autograder earned higher scores on both their initial and final submissions. They also submitted more than twice as often as those using the old autograder. Each of these differences was found to be statistically significant (p=0.0227, and 0.0002, and 0.0013, respectively).

We also see that old-autograder students on average performed worse on the links-based implementation than on the array-based implementation, while the opposite is true for new-autograder students.

6.3 Analysis

The first finding is that students provided with the new autograder earn a higher grade with their initial submission than those provided with the old autograder. This difference cannot be explained by the quality of the feedback provided, as this result is occurring before students have seen any feedback. One might try to explain it by imagining that the new autograder is less thorough or more generous than the old one, but this is not the case. Both autograders test the same scenarios, assigning them the same weights.

Rather, the difference is that the new autograder is more precise than the old autograder. Suppose that a student has written a broken implementation of the methods for adding to a sequence, but a correct implementation of the method for removing from a sequence. The old autograder will not be able to detect that the removal method works, because it sets up the environment for its tests using the student's broken adding methods. The new autograder will give the student full credit for all of the removal tests because it does not rely on the student's broken adding methods to set up those scenarios.

The second finding is that final grades are also higher for new-autograder students than old-autograder students. This can be explained both by the new autograder's ability to bypass broken student code when testing working student code and by the new autograder's ability to better explain how and why student code is broken so that it can be fixed. Manual grading of all final submissions reveals that the latter explanation predominates.

An interesting side observation is that students using the old autograder received similar grades for their initial submissions on the array implementation and the links implementation, but were able to improve their grades more substantially on the array implementation than the links implementation. This is because the style of tests used by the old autograder is sufficient for observing all details of the array version's internal state. For the array implementation, `"A C (E) F"` tells us that the size is 4, the index of current element is 2, and the first four locations in the array are filled with those characters. For the links implementation it tells us that the head and cursor references and size are correct, but we get no information about the status of the tail or precursor. Thus, errors in setting these can be undetected and propagate from one method to another, hiding their true cause.

Finally, we note that students with the new autograder make far more submissions than those with the old autograder. There are several plausible explanations for this, but we believe the disparity in final grades points to the correct interpretation: students using the old autograder are more likely to give up, as the feedback from the autograder is not sufficient for them to understand their code's deficiencies, while students using the new autograder are encouraged to complete the project successfully because they are getting useful feedback with each submission.

6.4 Cautions

It is highly likely that students who were using different autograders communicated with each other and shared discoveries. It is certain that they received similar assistance from the instructor and tutors. To the extent that this affected the results, however, it would only artificially narrow the observed differences.

This evaluation demonstrates that students using the new autograder earn higher grades than those using the old autograder, but this does not necessarily imply that students using the new autograder have understood the underlying concepts more deeply or better achieved any other course outcomes. There are trade-offs in providing a "better" autograder, as students rely more on it than themselves to find and diagnose errors in their code.

We have found the techniques described here to work very well for certain types of assignments that show up in CS2: those in which we want students to practice working with a specific design of our choosing to build a module with important, tricky private implementation details. The linked version of Sequence exemplifies this property, and we have found a number of other CS2 assignments do as well. We do not use, and do not recommend the use of these technique for assignments in which there are no tricky private implementation details (because it would be unnecessary) or those in which students should be creating their own designs (because it would be unacceptably restrictive) or those in which students should be practicing their testing and debugging skills

(because it takes those responsibilities from them). Attempts to apply these techniques to all assignments for a course, or to all courses, are likely to cause more problems than they solve.

Rigidly defining the specification of a module down to even its private implementation details has a downside beside the obvious one of denying students the opportunity to practice design: correct solutions are naturally going to look similar to each other, which makes plagiarism detection more difficult. Instructors who do this will need to rely on other techniques for discouraging and detecting plagiarism. In our case, looking at the history of submissions is usually sufficient to determine whether or not similar-looking final solutions were independently derived.

7 RELATED WORK

The ability to expose autograders to students requires a framework for accepting submissions, running those autograders, and displaying feedback. While our experiments used AutoLab, there are many similar competing frameworks [2]. (Confusingly, many authors use the term "autograder" to refer to the framework, rather than the assignment-specific tests run by the framework.)

This paper has been about a technique for setting up and interrogating the internal state of the object on which a method is called. If a test instead needs to interrogate the state of another object the method interacts with, such as once of its parameters, reflection on that other object can be similarly effective. But if the test needs to be notified when events occur on those other objects, or needs to simulate behavior from those objects that would otherwise be impossible, the appropriate tool is a mock object. This can be as simple as manually writing a new class to the same interface as the one being mocked, but there are also frameworks for automatically building mocks, such as Mockito [12].

The *Tester* library for *ProfessorJ* uses reflection to automatically build `equals` and `toString` methods for arbitrary classes. These are used so that novice students can write unit tests checking equality of objects without needing to know how to implement these methods manually. When the student-written test asserts that objects should be equal, the reflection-based method actually performs the check. Thus, this is a tool to help students write their own tests, not to automatically grade student work [11].

If class invariants, method preconditions, and method postconditions are specified in a formal language such as JML, it can be possible to automate the production of unit tests for those methods. One technique for doing so does not rely on the type of reflection described in this paper but instead on injecting new methods directly into the class being tested to wrap the methods under test. The wrappers confirm that invariants and preconditions hold, call the underlying method, and then confirm that invariants and postconditions hold. Because these methods are injected into the original class, they automatically have access to private components and can interrogate the entire state [3].

A number of authors have proposed other ways in which reflection could improve or simplify autograding in different ways than our approach. One system uses reflection to find and call the methods a student wrote based on their signatures without knowing precisely what they are named, but does not use it for interrogating state. It avoids the problem of a test relying on other incorrect methods for setup by overriding those methods with correct versions while setting up a test scenario [9]. Another system known as AutoGrader compiles all student submissions together and uses reflection to create instances of each student's classes as they are ready to be tested [5]. Wilcox proposes a variety of testing strategies, including the use of reflection for determining whether or not class components exist and have the correct properties [13]. All of these ideas seem compatible with and complementary to our technique of interrogating and modifying state.

This paper has been about ensuring that unit test failures lead to precise, meaningful failure messages for novice students. There has also been much research into providing precise, meaningful syntax error and compiler warning messages for novice students [1].

A workshop at SIGCSE 2022 discussed a wide variety of best practices for writing autograders for novice assignments, and included a small section on the reflection-based ideas described in much more detail and evaluated here [6].

8 CONCLUSION

Automatically and exhaustively testing student code makes it easier to assign partial credit and quickly provide detailed feedback about deficiencies, either on a continual basis if exposing tests to students through an autograder or as part of the grading process if students submit untested (except by themselves) files. Writing exhaustive tests can be challenging, because well-designed classes do not expose to the public details that would be needed to confirm that class invariants are being maintained. The relatively static and prescribed nature of student assignments makes it reasonable to test these things when it would be inappropriate to do so on production code.

Reflection makes it possible to write precise tests in which setting up a scenario does not depend on correctness of any student code, and the results of actions can be fully observed even for private implementation details. Further use of reflection allows instructors to confirm that students are using only the state variables they have been asked to use and to conditionally compile and run tests only when the features they are designed to test exist.

This paper showed how to accomplish this using Java, which has mature and feature-rich libraries for unit testing and reflection. Similar techniques can be used in C#. C++ does not natively support reflection, but these ideas can be implemented through typecasts between pointers to objects from student classes and those from equivalent instructor classes. Reflection is also not needed to implement these ideas in languages such as C and Python that do not enforce encapsulation

We believe that tests that clearly and precisely identify the parts of a class that do not meet their specifications should have clear benefits for students and instructors compared to those that do not: students should earn higher grades on their assignments, should satisfy more functional requirements, and should learn more by getting more detailed feedback, while instructors should find that they spend less time debugging student code. A small experimental study has validated the most easily measurable one of these outcomes: student final grades are higher with reflection-based autograders than with traditional methods.

REFERENCES

[1] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland, UK). 177–210.

[2] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 2022. Stop Reinventing the Wheel! Promoting Community Software in Computing Education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland). 261–292.

[3] Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (Malaga, Spain). Springer-Verlag, 231–255.

[4] Steve Freeman and Nat Pryce. 2009. *Growing Object-Oriented Software, Guided By Tests* (1st ed.). Addison-Wesley Professional.

[5] Michael T. Helmick. 2007. Interface-based Programming Assignments and Automatic Grading of Java Programs. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland, UK). 63–67.

[6] Chad Hogg and Maria Jump. 2022. Designing Autograders for Novice Programmers. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education, V. 2* (Providence, RI, USA). 1200.

[7] Pattie Maes. 1987. Concepts and experiments in computational reflection. *SIGPLAN Not.* 22, 12 (1987), 147–155.

[8] Michael Main. 2011. *Data Structures and Other Objects Using Java* (4th ed.). Prentice Hall Press.

[9] Derek S. Morris. 2002. Automatically Grading Java Programming Assignments via Reflection, Inheritance, and Regular Expressions. In *Proceedings of the 32nd Annual Frontiers in Education Conference* (Boston, MA, USA). T3G–22.

[10] David O'Hallaran, Gregory Kesden, and et. al. 2014. AutoLab Project. https://autolabproject.com/.

[11] Viera K. Proulx and Weston Jossey. 2009. Unit Test Support for Java via Reflection and Annotations. In *Proceedings of the 7th International Conference on Principles and Practices of Programming in Java* (Calgary, Alberta, Canada). Association for Computing Machinery, 49–56.

[12] David Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock Objects for Testing Java Systems. *Empirical Software Engineering* 24, 3 (2019), 1461–1498.

[13] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education* (Memphis, TN, USA). 437–442.