REFLECTION-BASED PRECISE AUTO-GRADING

Chad Hogg
Millersville University
chad.hogg@millersville.edu

ABSTRACT
Automatic testing of student code is frequently used in programming-intensive computer science courses. However, typical techniques for writing tests cannot generate detailed and sensible feedback about the cause of a test failure, which introductory students need. This paper presents a technique using reflection to generate suites of tests for quickly determining the cause of every failure in a student-written class, provided that the design of the class is exhaustively specified.

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 enrollments increase 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 auto-grading system so that they can follow a submit - improve - resubmit workflow.

Unit testing suites such as JUnit provide a quick and easy way to write test cases that are good at discovering deficiencies. When a professional developer encounters a failed test, she 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, she can attach a debugger to the test and step through it to determine exactly where and when anomalous behavior begins. This is not true for a student submitting their assignment to an auto-grading 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 discovering the root causes of deficiencies.

A student submitting their code to an auto-grading system does not have direct access to the test case, only to a report of whether the test fails or succeeds. They will only get meaningful feedback about the failure if the designer of the test has built that feedback into it. 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 and plan for 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, particularly in introductory courses, are often fully specified and require students to solve problems in very specific ways. As such, it is not unreasonable to require that every student's class use the same fields, with the same names and types, and the same methods, with the same names and signatures. This means that reflection can 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 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 [1]. The contribution of this paper is to explain why this technique can be appropriate for testing introductory student assignments, and to provide a framework for doing so in Java.

2. An Example Assignment: Linked Sequence

We will be using an assignment from a CS2 course as an example of this technique. In it, students implement the

`Sequence` abstract data type, as described by Michael Main [2]. 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 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.

In this assignment the student implementation must use a basic 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`. Students are provided the following class invariants:

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. Because most 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.

3. Traditional Tests

The simplest tests for this class would construct an object, 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, 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 an object, manipulate that object to get it into the state that leads to a specific scenario, 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 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.

```
1  @Test
2  public void testAddAfterNoCurrent() {
3    LinkedSequence<Integer> sequence =
4      new LinkedSequence<>();
5    sequence.addAfter(5); // "(5)"
6    sequence.addAfter(3); // "5 (3)"
7    sequence.advance();   // "5 3"
8
9    sequence.addAfter(2); // "5 3 (2)"
10
11   assertEquals("...", "5 3 (2)",
12     sequence.toString());
13   assertTrue("...",
14     sequence.hasCurrent());
15   assertEquals("...", 2,
16     sequence.getCurrent());
17   assertEquals("...", 3,
18     sequence.size());
19 }
```

Figure 1: A poor quality test.

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.

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 public fields would encourage poor design, so it must be avoided. Fortunately, Java offers a library for reflection, with which we can reason about and directly modify the structure of our 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.

Figure 2 shows how to give yourself access to a field, even if it is private. Figures 3 and 4 show how to then read and write that field, respectively. In these, `object` 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.

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

Figure 2: Making a field accessible.

```
1 Node<E> head =
2   (Node<E>)headField.get(object);
```

Figure 3: Reading the value of a field.

```
1 headField.set(object, value);
```

Figure 4: Writing the value of a field.

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. Reflection-Based Tests

Figure 5 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 20 call the method that we are testing – the only `LinkedSequence` method call that exists anywhere in this test. Lines 22 through 35 confirm that the sequence's `head` refers to the beginning of a linked list containing 5, 3, and 2. Lines 36 and 37 confirm that the first two nodes of the list are the same objects they had been. Lines 38 through 45 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.

One feature of this test that may seem odd is the use of `assertFalse()` / `assertTrue()` where `assertNotNull()` / `assertNull()` / `assertEquals()`, or even Hamcrest-style assertions

```
1  @Test
2  public void testAddAfterNoCurrent() {
3    Integer[] elements = {5, 3};
4    Node<E> list = makeList(elements);
5    Node<E> first = list;
6    Node<E> second = first.getNext();
7    LinkedSequence<Integer> sequence =
8      new LinkedSequence<>();
9    R.setHead(sequence, first);
10   R.setTail(sequence, second);
11   R.setCursor(sequence, null);
12   R.setPrecursor(sequence, null);
13   R.setSize(2);
14
15   try {
16     sequence.addAfter(2);
17   }
18   catch(Exception exception) {
19     fail("..." + exception);
20   }
21
22   Node<E> node1 = R.getHead(sequence);
23   assertFalse("...", null == node1);
24   Node<E> node2 = node1.getNext();
25   assertFalse("...", null == node2);
26   Node<E> node3 = node2.getNext();
27   assertFalse("...", null == node3);
28   assertTrue("...", null ==
29     node3.getNext());
30   assertTrue("...", 5 ==
31     node1.getValue());
32   assertTrue("...", 3 ==
33     node2.getValue());
34   assertTrue("...", 2 ==
35     node3.getValue());
36   assertTrue("...", first == node1);
37   assertTrue("...", second == node2);
38   assertTrue("..." node3 ==
39     R.getTail(sequence));
40   assertTrue("...", node3 ==
41     R.getCursor(sequence));
42   assertTrue("...", node2 ==
43     R.getPrecursor(sequence));
44   assertTrue("...", 3 ==
45     R.getSize(sequence));
46 }
```

Figure 5: A high quality test.

would seem more appropriate. When writing tests for experienced developers who will have access to the source code for the tests themselves, the extra information from those 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 should 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 23, 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".

Writing the test of Figure 5 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.

6. Testing Structure

The test shown in Figure 5 does not make many assumptions, but it does assume the existence of fields named `head`, `tail`, `cursor`, `precursor`, and `size` with the correct types, as well as the existence of a method named `addAfter` with the correct signature. If the method does not exist, the test will not compile. If a field does not exist, the reflection-based setup for the test will throw a `NoSuchFieldException`.

These problems can be avoided by writing an additional test that simply confirms that the `LinkedSequence` class has all required components. It can also confirm that the class has no additional components.

Figure 6 shows a test for confirming that a class contains a `head` field and no others. To confirm that it has each of five fields, it can be naturally extended with four more analogous cases. The process for confirming that methods with particular signatures exist is sufficiently similar that we are not showing it. We refer to these types of tests as *structure* tests, and those like Figure 5 as *outcome* tests. With a custom test harness it is possible to run structure tests first and skip the outcome tests if any of them fail, thus avoiding showing any confusing error messages to students.

```
@Test
public void testFieldDeclarations() {
  Field[] fields = LinkedSequence.
    class.getDeclaredFields();
  boolean foundHead = false;
  for(Field fld : fields) {
    switch(fld.getName()) {
      case "head":
      foundHead = true;
      assertTrue("...",
        Modifier.isPrivate(
          fld.getModifiers()));
      assertFalse("...",
        Modifier.isStatic(
          fld.getModifiers()));
      assertSame("...", Node.class,
        fld.getType());
      break;
      default:
      fail("..."); // extra field
  }
  assertTrue("...", foundHead);
}
```

Figure 6: A structure test.

7. Testing Partially-Written Classes

In the previous section we mentioned that if a test tries to call a method that does not exist in the class, the test will not compile. This is not a potential issue in the `LinkedSequence` example, because all methods students must write are part of an interface that the class implements, so the student class would also not compile unless all of them exist.

But it is also possible, especially in a first semester programming course, to design an assignment in which students must eventually write several independent methods. If there is no interface or abstract superclass requiring that they be implemented, the student's work will compile when they have only written some of the methods. It would be nice if a partially-written class could also be tested, so that students can confirm that earlier work was correct before moving on, and so that partial credit can be given for incomplete assignments. If test classes will not compile because they call methods that do not exist, this will be impossible.

Reflection also makes it possible to work around this. To do so, divide the tests into several classes. Each method the student must write gets one class containing structure tests and one containing outcome tests. The structure tests must confirm that all class components used by the outcome tests exist and have the correct format. Structure tests will always compile, and will simply fail (with custom, meaningful messages such as "The LinkedSequence class should have contained a field named head") if something they are looking for is missing. Outcome tests will not compile if a method they wish to call is missing, but in this case we will avoid trying to compile them.

A custom test harness can, for each method to be tested, first compile and run the structure tests. If the structure tests pass, it then compiles and runs the outcome tests. Otherwise, it neither compiles nor runs them but counts them as if they had all failed. In order to make compilation of a source optional, code should not directly reference its class but only do so through reflection. This code is too complicated to show compactly, but it uses `javax.tools.JavaCompiler`'s `run()` method when you have decided to compile a class that was not linked into the project and `org.junit.runner.JUnitCore`'s `run` method to execute the tests in a class that was not linked into the project.

Using these tools an auto-grader can give a student credit for having completed the first two parts of an assignment, and constructive feedback about them, even when the remaining parts do not yet exist.

8. Related Work

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 [3].

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 [4].

If class invariants, method preconditions, and method post-conditions 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 pre-conditions hold, call the underlying method, and then confirm that invariants and post-conditions hold. Because these methods are injected into the original class, they automatically have access to private components and can interrogate the entire state [5].

Another 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 [6].

This paper has been about ensuring that unit test failures lead to precise, meaningful failure messages for novice students. There has also been a great deal of research into providing precise, meaningful syntax error and compiler warning messages for novice students [7].

9. 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 auto-grader 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. Other languages have varying levels of support for these features. For example, the standard library for C++ contains nothing on either of these topics, but the third-party library Catch makes writing unit tests relatively painless and the third-party library RTTR provides reflection capabilities. As future work, we intend to build equivalent infrastructure for other programming languages frequently used in introductory courses.

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. We have not yet had an opportunity to verify this theory but intend to gather data this semester by having one section of students use the new tests (like Figure 5) and another the old ones (like Figure 1).

References

[1] S. Freeman and N. Pryce, *Growing object-oriented software, guided by tests* (Boston, MA: Addison-Wesley, 2010).

[2] M. Main, *Data structures and other objects using Java, 4th edition* (Upper Saddle River, NJ: Pearson, 2012).

[3] D. Spadini, M. Aniche, M. Bruntink, & A. Bacchelli, Mock objects for testing Java systems, *Empirical Software Engineering, 24*(3), 2019, 1461-1498.

[4] V. K. Proulx and W. Jossey, Unit test support for Java via reflection and annotations. *Proc. 7th Intl. Conf. on Principles and Practice in Java*, Calgary, Alberta, Canada, 2009, 49-56.

[5] Y. Cheon and G. T. Leavens, The JML and JUnit way of unit testing and its implementation. *Proc. 16th European Conf. on Object-Oriented Programming*, Malaga, Spain, 2002, 231-255.

[6] D. S. Morris, Automatically grading Java programming assignments via reflection, inheritance, and regular expressions. *Proc. 32nd Frontiers in Education Conf.*, Boston, MA, 2002, T3G-22.

[7] B. Becker, et. al., Compiler error messages considered unhelpful: the landscape of text-based programming error message research. *2019 ITiCSE Working Group Reports*, Aberdeen, Scotland, UK, 2019, 177-210.