Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 2006

Interface-based Programming Assignments and Automatic Grading of Java Programs

Michael T. Helmick
Miami University

Interface-based Programming Assignments and Automatic Grading of Java Programs

Michael T Helmick
Miami University
Department of Computer Science and Systems Analysis
Oxford, Ohio USA
mike.helmick@muohio.edu

ABSTRACT

AutoGrader is a framework developed at Miami University for the automatic grading of student programming assignments written in the Java programming language. *AutoGrader* leverages the abstract concept of interfaces, brought out by the Java `interface` language construct, in both the assignment and grading of programming assignments. The use of interfaces reinforces the role of procedural abstraction in object-oriented programming and allows for a common API to all student code. This common API then enables automatic grading of program functionality. *AutoGrader* provides a simple instructor API and enables the automatic testing of student code through the Java language features of *interfaces* and *reflection*[1]. *AutoGrader* also supports static code analysis using PMD [4] to detect possible bugs, dead code, suboptimal, and overcomplicated code. While *AutoGrader* is written in and only handles Java programs, this style of automated grading is adaptable to any language that supports (or can mimic) named interfaces and/or abstract functions and that also supports runtime reflection.

1. INTRODUCTION

Automatic grading of student programs has been of interest to computer science educators for some time [8] and continues to gain attention today [7, 5, 9, 12, 14, 10]. There have been various approaches with most leaning towards analyzing the output of the entire program with some mention of techniques and tools such as JUnit [16]. However, there seems to be no established consensus on the best way to automatically grade student code.

We take the position that there are two tasks involved in grading of student assignments:

1. Functional testing
2. Evaluation of design and style

[1]The Java reflection API allows the runtime system to be inspected and manipulated dynamically. `http://java.sun.com/docs/books/tutorial/reflect/index.html`

Preliminary analysis of Java coding style can be handled automatically using either the open source Checkstyle [1] or PMD [4] packages (*AutoGrader* is integrated with PMD). While it might be possible to handle the evaluation of program design in an automated fashion, we feel that this task is best left to an experienced instructor or teaching assistant. There is a large body of program design knowledge that comes from experience, and this experience can not easily be programmed into an automated system.

Functional testing, on the other hand, is a task that produces a boolean answer. A program either works and provides the correct answer based on predetermined inputs, or it does not. Since software systems tend to be built in layers, we can further sub-divide assignments and target specific functionalities for testing. Our goal in automatically testing student code is two-fold: (1) to adequately exercise student code; and (2) to segment the tests such that pieces of functionality are isolated and tested as independently as possible. This not only makes automatic assessment easier, it is also a valuable teaching technique for instruction of software construction.

We have introduced a pedagogy for our data structures course that enables and utilizes automatic grading for programming assignment functionality. Interface-based programming can raise the level of abstraction within a program and is often underutilized [15]. The use of interfaces with the Java programming languages enforces an extra level of indirection and allows the instructors to set a certain level of commonality in a *what is to be done* manner, without regard to the individualized implementation details. While seeming restrictive on the surface, this still leaves implementation considerations up to individual students and does not impose a common design, just a common interface. This also unifies grading of student work since each individual solution adheres to a prescribed interface, allowing the instructor to save time by constructing the tests a single time and applying them to all students in the course.

This paper describes how we use interfaces in our data structures class, how this enables discrete functional testing of student code, and how this is integrated into the *AutoGrader* framework. Section 2 describes how interfaces are used in the data structures curriculum. Section 3 gives an overview of the *AutoGrader* framework, how it is constructed, and how grading packages are written by an instructor. Section 4 places this framework in the context of related work in this field.

2. INTERFACE-BASED ASSIGNMENTS

For programming assignments in the context of a data structures course, there is a certain level of commonality between all student projects. When talking about abstract data structures there is a natural tendency towards using a common interface. For instance, data structures for lists can have multiple implementations: array-backed list, singly linked list, and doubly linked list. It doesn't make sense that each of these implementations would present a different interface to the clients of list services. In fact, the usefulness of these data structures increases if they implement common interface[11], emphasizing the flexibility of switching implementations as long as the client only utilizes the interface. Presenting data structures in this fashion is a worthwhile educational objective [15].

We emphasize the use of the Java language construct of `interface` in order to increase student awareness of component software and to show that we can test software without knowing the implementation details. *Black box testing* is the process of testing software where we craft tests based solely on the interface of the software, ignoring what we know about the implementation details. For the list assignment in our data structures course, we assign students the task of implementing the standard `java.util.List<E>` interface [2] by providing both array-backed and linked list implementations. Because of the complexity involved, the `subList` method is excluded.

Students are also required to write unit tests using JUnit [3] for all aspects of their programs (*White box testing*). This serves two purposes: (1) to get students to test their code; (2) again, to emphasize the use of interfaces in procedural abstraction, getting them to focus on *what* is being done, rather than *how* it is being done. If tests are properly written against the `List<E>` interface and not directed towards a specific implementation, they are able to reuse their tests for both list implementations, reducing their overall time spent on the assignment.

Configuring a JUnit test class to be able to use multiple implementations can be enabled through the use of an abstract class containing all of the tests, and two sub-classes containing the code to instantiate the list classes. This separation further emphasizes good unit testing in that it makes it more difficult to make one test dependent on another test in the test package. Students are provided with some starter code to demonstrate constructing tests in this way. An excerpt of this starter code is shown in Listings 1 and 2. This example setup works well when students are to provide more than one implementation of the same interface and also provides an example of how code can be tested at the interface level (Listing 1).

Listing 1: GeneralListTest.java

```
1  import java.util.List;
   import junit.framework.TestCase;
3
   public abstract class GeneralListTest
5      extends TestCase {

7    public abstract <T> List<T>
         getList( Class<T> clazz );
9
     /// test the constructor for a list
11   public void testList() {
       List<String> strList =
13         getList( String.class );

15     assertNotNull( strList );
       assertEquals( 0, strList.size() );
17   }

19 }
```

Listing 2: ArrayListTest.java

```
1  import java.util.List;
   import java.util.Random;
3
   public class ArrayListTest
5      extends GeneralListTest {

7    @Override
     public <T> List<T> getList(
9          Class<T> clazz) {
       return new studentid.ArrayList<T>();
11     //return new java.util.ArrayList<T>();
     }
13 }
```

The code from Listing 1 is then completed by the student, fully testing the list interface. Listing 2 shows how easy it is to then either test the student developed code (`new studentid.ArrayList<T>()`) or test the built in Java implementation (`new java.util.ArrayList<T>()`). This enables students to run their tests against an implementation of an object that implements the same interface and is previously known to be in full working order. While the ability to do this is extremely helpful to someone who is new to unit testing, it is not always possible to test an existing implementation. For assignments where there is not a built in data structure, an instructor provided implementation can be used as the reference implementation. On more advanced assignments, students are generally not provided with a working implementation to compare against.

Interfaced-based assignments have been used successfully in our data structures course for lists, binary trees, heaps, sets, maps, and graphs. We emphasize the portability of client code using an interface and thus the ability for a student to write one test suite which can then test multiple implementations. The use of assignments of this type and subsequent automatic grading of student code is not limited to the scope of a data structures course. Any course where the instructor can provide a common interface to students can be handled in this fashion and graded with *AutoGrader*.

3. AUTOMATED FUNCTIONAL TESTS

The *AutoGrader* framework is based on the concepts and operation of JUnit. It is geared towards the execution of student code where the implementation of a common interface is implemented as described in Section 2. *AutoGrader* itself is a lightweight framework consisting of only 17 source files and less than 1,500 lines of code, making it easily portable and available for integration with other utilities. In order to automatically grade a student assignment, there must be uniform class names across the assignment. For instance, all students will write a class called `ArrayList` with different implementations being designated by being included in a package name unique to each student. We

have found success in having each student create a package whose name is their unique university id, so that classes are named `studentid.ArrayList`.

An instructor must provide the actual tests to be run, a list of student IDs, and the name of the concrete classes to instantiate. There are three classes in *AutoGrader* that are used by an instructor when grading an assignment: *Graded* objects, *GradingSession* and *GradingPackage*.

Objects that extend `edu.muohio.csa.autograder.framework.Graded` are similar to objects that extend `TestCase` in JUnit. The major difference is that rather than instantiating the object to be tested directly, the object is located using the provided `getInstanceOfObject` method. An instructor would write an object extending `Graded` for each objective that they want to test. As in the example described earlier, an object is written to test a specific interface and can even be reused to test different implementations of the same interface for an assignment. Listing 3 shows the beginning of code actually used to exercise student code for the list assignment. Randomization is used within a certain boundary to further exercise the code.

Listing 3: ArrayListTest.java

```
import java.util.List;
import java.util.Random;

import edu.muohio.csa.autograder.
  framework.Graded;
import edu.muohio.csa.autograder.
  framework.GradingException;

public class ListGrader extends Graded {

  /// test add
  @SuppressWarnings("unchecked")
  public void grade_Add_Element()
    throws GradingException {

    int testSize =
      (new Random()).nextInt(100) + 15;

    List<Integer> intList = (List<Integer>)
      this.getInstanceOfObject( List.class );

    for( int i = 0; i < testSize; i++ ) {
      intList.add( new Integer(i) );
      assertEquals( new Integer( i+1 ),
          new Integer( intList.size() ) );
    }
  }

}
```

As a matter of convention, any method in a class extending `Graded` whose name begins with `grade` will be identified via reflection and considered to be a test method. Several utility methods are included in the `Graded` object base class to assist with test case construction, including:

1. `getInstanceOfObject(class, Object...)`: Returns an instance of the student code object under test (see example in Listing 3). The actual object instantiated is dependent on the current student being processed in the grading session. The first parameter indicates what class/interface the instantiated object should be returned as. This parameter also determines the return type of the method since `getInstanceOfObject` is a generic method. Optionally, a variable length list of parameters can be passed, allowing the framework to locate a specific constructor in the student's class using reflection. If the student did not provide an implementation of the required object for this test, that will be logged in the final report.

2. `assertEquals( expected, actual )`: An assert equals method that is overloaded for several different types, including `Object` so that all types of objects can be compared. If the assertion fails, `GradingException` is thrown and if caught by the framework will be logged in the report for this student.

3. `assertTrue()` and `assertFalse()`: Assert that a boolean expression evaluates to true (or false).

4. `fail(message)`: Force this test to be marked as failed immediately. This is useful when you expect code to thrown an exception, and it does not.

This framework would not be workable without the reflection capabilities built into the Java language. Reflection enables the code to be dynamic in nature making use of classes loaders to locate student code, and reflection to actually execute the test methods. To invoke each individual test, a reference to the current `Graded` object and current test `Method` are handed to a background invoker. This enables each individual instantiation of student code and execution of tests to be run in a separate thread, allowing for the thread to be terminated should it be found to run for too long. Early versions of *AutoGrader* did not do this, but infinite loops found in submitted student code required that there be a mechanism to stop testing of a long running method and to move on to other tests and students.

When invoking a test, all possible exceptions are caught and wrapped in a `GradingException` if the test case itself does not catch the exception. This allows for tests to be constructed that purposely induce exceptions on student code, and records a failure if the exception is not thrown.

After creating the various `Graded` objects for an assignment, they can be composed into grading packages, and ultimately a grading session. A `GradingPackage` is constructed by passing in an instance of the `Graded` object to use, a name to display on the grading output reports, and the name of the student classes to instantiate.

Multiple `GradingPackage` instances are added to a single `GradingSession` instance which is also configured with the output directory to use. An optional text user interface can be attached to the grading session which provides constant updates during the execution of the program, but provides a slight performance penalty. Listing 4 shows code actually used to construct the grading session for a programming assignment. This example displays the possibilities for reuse of the same test harnesses to test different implementations of the same interface. Each `GradingPackage` is a different section in a student's final report, a sample report is available in Appendix A.

Listing 4: GraderMain.java

```
GradingSession session =
    new GradingSession();
session.addTextUI();
session.addFileReporter(
    "/output/directory/");

// ADD the grading packages
session.addPackage( new GradingPackage(
    new ListGrader(), "ArrayList-Phase I",
    "ArrayList" ) );
session.addPackage( new GradingPackage(
    new ListGrader(), "LinkedList-Phase I",
    "LinkedList" ) );

// called once for each student
// or loaded from a file
session.addStudent(
  new StudentRecord( "studentid",
                     "Student Name" ) );
session.run();
```

Once the session object's `run` method is invoked, the tests are run by executing each grading package (in the order added) for each student (in the order added). The appropriate student class file is located using the default class loader, but the object is not yet instantiated. Each test is then run one at a time, with each invocation started in a separate thread and allowed to run up to the configurable timeout value. New instances of the student objects are created each time the `getInstanceOfObject` method is invoked within a test.

AutoGrader also makes use of PMD [4] to evaluate the style of student code. PMD checks for some common coding problems such as empty catch blocks, empty if statements, comparing objects with `==` instead of the `equals` method. A full list of Java style rules supported by PMD is available on the PMD Web site. We have found that PMD provides a good starting point for basic evaluation of coding style, but we still recommend manual evaluation of code by the instructor.

The *AutoGrader* framework is both powerful and flexible, and integrates with interface-based programming assignments to test student code for functionality and provide feedback reports. For each student assessed during a grading session, a text file is created reporting what the result is for each test. A passed test receives a simple notation of "passed", while a failed test indicates what comparison failed, or which line of code threw an exception during execution. Details of what is being tested are hidden from the student, leaving the option of disclosure to individual instructors. Descriptive test names convey the intent of the test, but we allow students to see the actual code of the tests upon request.

4. RELATED WORK

Various efforts for automatic grading of student programming assignments have been undertaken. Many systems rely on pattern matching techniques as is done in both ASSYST [10] and TRY [13]. The major drawback to this approach is only the final output of the program is checked for correctness against the final "correct" output. The *AutoGrader* framework allows for an instructor to take a narrower view of correctness and assess a student's performance on a method by method basis. Properly constructed test cases will explicitly test a method for correct or incorrect under a single circumstances, leading to the possibility of multiple tests for an individual method.

It has been suggested that introducing named interfaces earlier in the curriculum is worthwhile [15]. Named interfaces help students to learn the concept of procedure abstracting and the decoupling of what an object does from how it is implemented.

Some previous work has been done in testing of student code using the JUnit [3] framework and OCETJ [16]. As with *AutoGrader*, the authors note that this approach promotes functional testing rather than textual output comparison. OCETJ appears to exercise student code through JUnit, but details of how the testing is implemented and availability of a batch mode are not specified. The *AutoGrader* framework goes farther in providing a comprehensive framework that can be deployed in batch mode for the instructor and includes background invocations to catch long running student code. The flexibility of the *AutoGrader* code, including use of the observer pattern [6] makes integrating *AutoGrader* into your existing submission infrastructure a possibility. Elements of the framework extend the Java `Observable` class, allowing you to write custom code registering as an observer of grading events.

This work further differentiates itself from use of JUnit in OCETJ in terms of namespaces. If the same JUnit test cases are run on multiple students' code, that requires each student to construct their class in the same namespace. This, in turn, requires a new Java virtual machine instance to test each students code. In *AutoGrader*, we provide for separate namespaces (Java packages) to be used for each student in the class, allowing the tests to be run in batch mode and in the same Java virtual machine instance.

The *AutoGrader* framework is more lightweight than the full submission system such as Web-CAT [5]. Our framework is meant to be a solution that can be rapidly deployed and run in an offline fashion or be easily integrated into other online systems. For instance, *AutoGrader* is integrated with our departmental submission system, but is easily disconnected and can be used with other submission systems. Our solution to automated assessment has the same goals as the Web-CAT system, but takes a different approach.

5. CONCLUSIONS

AutoGrader is a framework for Java developed at Miami University for the automatic grading of programming assignments based on common interfaces. This framework has been used successfully for over a year in our data structures course to grade programming assignments for functionality. Grading test suites implemented thus far have ranged in size from 30 to over 100 test cases. Test cases themselves range in complexity in order to adequately test student code. A test suite containing 90 tests can be run for a course containing 50 students in just over 10 minutes, providing an effective use of time for the instructor. Student source code is (optionally) checked for style using PMD [4] and its included style rules.

The assignments are presented to students as a task of implementing an existing or instructor provided interface promoting procedural abstraction while making each student's code testable by the same set of tests. Automated grading of students' code requires an instructor to develop unit tests

aimed at verifying discrete segments of the assignment.

While the framework has been used for data structures assignments including lists, binary trees, heaps, sets, maps, and graphs, the framework can grade any code for which there is an interface and for which unit tests can be written. In the end, *AutoGrader* provides an assessment of only whether or not the program, or parts of the program, produce the correct answer. An instructor should still examine the students' code to provide advice and assessment of design and implementation style.

AutoGrader is being integrated into Computer Science CourseWare (CSCW), an integrated online courseware solution developed at Miami University, and will enable students to receive feedback on their assignments before final submission of their solution. The complete source code for the *AutoGrader* framework is available for download at `http://www.users.muohio.edu/helmicmt/autograder/`. Work is also underway to integrate Java code coverage analysis of student written unit tests.

6. REFERENCES

[1] http://checkstyle.sourceforge.net/.

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/util/list.html.

[3] http://junit.org/index.htm.

[4] http://pmd.sourceforge.net/.

[5] S. H. Edwards. Teaching software testing: automatic grading meets test-first coding. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–319, New York, NY, USA, 2003. ACM Press.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] J. A. Harris, E. S. Adams, and N. L. Harris. Making program grading easier: but not totally automatic. *J. Comput. Small Coll.*, 20(1):248–261, 2004.

[8] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.

[9] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for coursemaster. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 46–50, New York, NY, USA, 2002. ACM Press.

[10] D. Jackson and M. Usher. Grading student programs using assyst. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, New York, NY, USA, 1997. ACM Press.

[11] J. Krone. Multiple implementations for component based software using java interfaces. *J. Comput. Small Coll.*, 19(1):30–38, 2003.

[12] L. Malmi, A. Korhonen, and R. Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 55–59, New York, NY, USA, 2002. ACM Press.

[13] K. A. Reek. The try system -or- how to avoid testing student programs. In *SIGCSE '89: Proceedings of the twentieth SIGCSE technical symposium on Computer science education*, pages 112–116, New York, NY, USA, 1989. ACM Press.

[14] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 133–136, New York, NY, USA, 2001. ACM Press.

[15] A. Schmolitzky. "objects first, interfaces next" or interfaces before inheritance. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 64–67, New York, NY, USA, 2004. ACM Press.

[16] G. Tremblay and E. Labonte. Semi-automatic marking of java programs using junit. In *International Conference on Education and Information Systems: Technologies and Applications (EISTA '03)*, pages 42–47, 2003.

APPENDIX

A. SAMPLE REPORT

Below is an excerpt from an AutoGrader report for a data structures assignment on graphs. The PMD style analysis is not shown.

```
** Test Package = Phase III (Shortest Paths)
  class tested: studentid.NamedGraph
  >>>>>>>>>>>>>> test 1 <<<<<<<<<<<<<<
     test name: gradeShortestPath
  running time: 0.101 seconds
   ** PASSED **

  >>>>>>>>>>>>>> test 2 <<<<<<<<<<<<<<
     test name: gradeShortestPath_2
  running time: 0.101 seconds
   !! FAILED !!
   reason=java.lang.NullPointerException
 location=edu.muohio.csa.autograder.Invoker.run(
Invoker.java:35 )
            java.lang.Thread.run(Thread.java:613)

  >>>>>>>>>>>>>> test 3 <<<<<<<<<<<<<<
     test name: gradeShortestPath_3
  running time: 0.103 seconds
   ** PASSED **

  >>>>>>>>>>>>>> test 4 <<<<<<<<<<<<<<
     test name: gradeShortestPath_4
  running time: 1.100 seconds
   ** PASSED **

  >>>>>>>>>>>>>> test 5 <<<<<<<<<<<<<<
     test name: gradeShortestPath_5
  running time: 0.101 seconds
   ** PASSED **

** package results: 4/5   pass percentage: 80.00
** End Test Package = Phase III (Shortest Paths)
```