Testing Strategies for the Automated Grading of Student Programs

Chris Wilcox
Computer Science
Colorado State University
1873 Campus Delivery
Fort Collins, Colorado 80523
wilcox@cs.colostate.edu

ABSTRACT

Enrollments in introductory computer science courses are growing rapidly, thereby taxing scarce teaching resources and motivating the increased use of automated tools for program grading. Such tools commonly rely on regression testing methods from industry. However, the goals of automated grading differ from those of testing for software production. In academia, a primary motivation for testing is to provide timely and accurate feedback to students so that they can understand and fix defects in their programs. Testing strategies for program grading are therefore distinct from those of traditional software testing. This paper enumerates and describes a number of testing strategies that improve the quality of feedback for different types of programming assignments.

CCS Concepts

•Social and professional topics → Computing education; Student assessment;

Keywords

automated assessment; automated grading

1. INTRODUCTION

Automated grading addresses some of the logistical challenges posed by increasing enrollments in introductory computer science courses. More importantly, automated grading gives students immediate feedback on their programs, allowing them to make progress without an instructor or teaching assistant by their side. The use of automated grading is widespread and generally thought to be beneficial [1]. We make the further claim that the use of grading systems has become an economic necessity for many institutions.

During our paper presentation at SIGCSE 2015, we were able to survey attendees concerning their use of automated assessment. Because of the venue, our survey risks a bias in favor of committed users of automated grading. Even so, the results provide a first approximation of tool usage. Figure 1 shows a summary of the responses in which it appears that the majority of attendees that use automation often or extensively also use homegrown tools. Such a high usage of homegrown tools motivates the need to study and disseminate effective testing strategies.

Many papers have presented frameworks for automated grading, including Web-CAT [2], Marmoset [6], APOGEE [4], and Infandango [5]. More recently, commercial software has become available for online grading, including Vocareum, Zybooks, Turings Craft, and Pearson Revel. Such software is commonly based on regression testing, a technique widely used for quality control in industry. However, we maintain that the goals of program grading differ significantly from those of software production.

In previous work we find evidence that automated grading contributes to student achievement and frees up teaching resources for better uses [7]. This paper puts aside the question of pedagogical benefit, and focuses instead on the pragmatic concern of how to improve the feedback from automated grading, using a variety of strategies that have not to our knowledge previously appeared in the literature. These strategies allow instructors to tailor tests to a wide range of programming assignments.

The goal of regression testing as practiced in industry is to verify that code changes do not introduce new defects. Developers initially create a test suite, then they write software that can pass some or all of the tests. When the test suite is run, the results can be archived and checked for correctness. Test suites are then run periodically during further development (or maintenance) of the software to identify defects that may have been introduced.

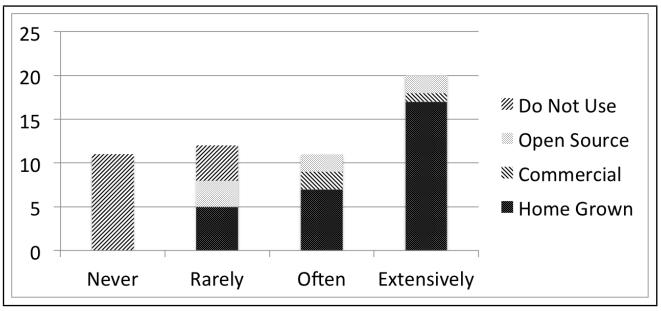

Figure 1: Automated Grading Survey Results.

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA

DOI: http://dx.doi.org/10.1145/2839509.2844616

Contrast this with the process for automatic grading of student programs, in which the instructor specifies an assignment, develops a solution, and creates a test suite. As with regression testing, test results from the solution are archived for use as an oracle. Student programs are submitted and tested by executing the test suite on them and comparing the outcome to the archived results. A mismatch with the oracle implies an error in the student program.

One of the key challenges of program grading is designing feedback to the student to help them understand defects in their program. We start this topic with an overview of grading frameworks (Section 2), followed by our central discussion of strategies for writing effective test programs (Section 3). We conclude with a discussion of several issues that arise in the context of grading frameworks (Section 4).

2. GRADING FRAMEWORKS

The key features of grading frameworks are 1) to provide feedback to the student about errors in their programs, 2) to automate the rote task of running student programs and scoring the results, and 3) to help instructors write effective test programs. Of equal importance, in our opinion, is a repository of test programs that can be leveraged for new assignments, thereby avoiding the necessity of developing unique test code for each assignment.

As with many institutions, we fulfill these requirements with a homegrown grading framework that provides a high degree of flexibility when developing test programs. The back-end of our framework is a set of shell scripts that automate the batch grading of student programs. A front-end web interface allows the instructor to offer real-time, interactive testing. The combination of script-based and web-based technologies is a third-generation system, as defined in the review of automated assessment by Douce [1]. Figure 2 shows the user interface and Figure 3 a block diagram of our grading system, which has previously been described [7].

Our automated tools rely on test scripts and program written by the instructor. These scripts specify test names, a scoring rubric, and a command for each test case. To support the range of testing strategies described in this paper, our grading framework allows the instructor to specify the entire Linux command line for each test case. A benefit of command line approach is that any programming language for which the platform has support can be tested. The command lines are executed in a Linux shell, and therefore can combine Linux commands and scripts, test programs, and any other software that runs in the Linux environment.

Figure 2: Automated Grading Interface.

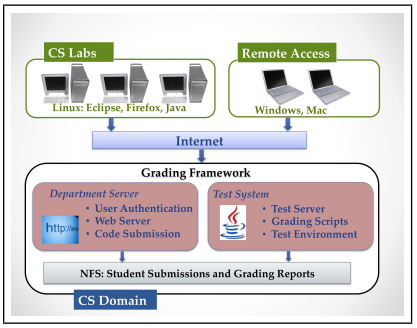

Figure 3: Automated Grading Framework.

3. TESTING STRATEGIES

In this section we focus on a number of strategies for the automated grading of student programs that have not to our knowledge been previously enumerated in the literature. These depend on a robust and flexible automated grading framework. To handle diverse assignments we employ eight strategies, listed below by increasing complexity.

3.1 Output Comparison

Output Comparison is similar to regression testing as practiced in industry. The strategy compares textual output from student programs to an oracle created from a correct solution to the assignment. Output can be captured from the standard output or error streams, and our framework allows the instructor to select a subset of the output lines to be compared. Our tests can restrict the capture to the granularity of a method call, using a Java feature that allows redirection of console streams.

To implement the strategy, our framework compiles and runs the student program, compares the resulting output to the oracle, and produces a grading report. The report contains comparison results, test scores, and compiler errors and warnings from the student code. Figure 4 shows a comparison in which the student submitted a program with a spelling error, a wrong answer, and extra output.

Frameworks assign scores based on the extent to which the student's output matches that of the oracle. A direct comparison is inflexible about capitalization and spelling errors and minor white space differences. For this reason, some frameworks allow customization of the comparison tool, which can be as rudimentary as a call to the Linux *diff* command, or can incorporate more advanced features such as color coding. Output comparison is primarily for console programs, and does not support feedback beyond direct program output. However, no separate test program needs to be written to use this strategy. Figure 5 shows a block diagram of the output comparison strategy.

Figure 4: Test Results from Output Comparison.

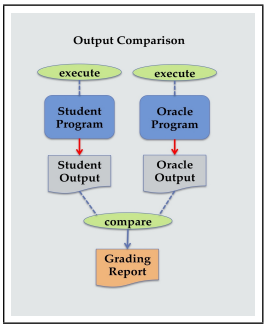

Figure 5: Output Comparison Strategy.

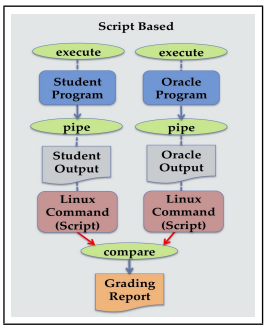

Figure 6: Script Based Strategy.

3.2 Script Based

Script Based testing extends the previous strategy by letting instructors specify a command line for each test, thereby introducing the use of Linux commands, scripts, redirection, pipes, and other shell programming features. We commonly pipe the textual output of student programs to Linux commands such as *grep*, *awk*, and *sed*, and we introduce custom shell scripts into the command line to post-process program output, which allows us to filter, sort, and otherwise interpret student results.

For example, a test case can use the Linux commands *sort* to sort, *grep* to filter, and *head* or *tail* to select output. Figure 6 shows a block diagram of script based testing in which the program output is piped to a single Linux command. However, the command line can invoke an unlimited number of commands and scripts, thus making the technique more powerful. Additional feedback can be incorporated by echoing text into the output stream.

3.3 Output Analysis

Output Analysis allows us to add custom test programs to the Linux command line to further analyze or post-process the output of the student program. We do not include in this strategy test programs that instantiate student classes or execute student programs as child processes. Such programs are described in subsequent sections. To implement output analysis, the test script pipes the student output to the test program, which then produces output for the comparison. As with the previous strategy, the instructor can incorporate Linux commands and scripts or other third-party software to help with the desired analysis.

An advantage of this approach is that instructors can write more sophisticated test programs than can be easily encapsulated into a Linux script. Test programs can be written

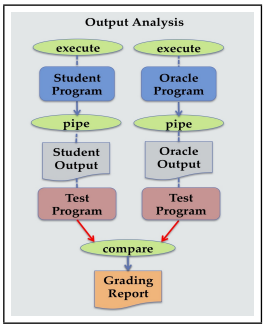

Figure 7: Output Analysis Strategy.

in any language to verify student programs in any other language. For example, the instructor can write a Perl or Python script to evaluate Java, C, or C++ programs. Specifying test scripts as a set of Linux command lines has the additional benefit of removing any restriction on the programming language used to write test code. Additional feedback can be inserted at any point by the test program to further explain the test results to the student.

Furthermore, the test program can do more than simply manipulate the program output. It can interpret the output and derive its own conclusions about the behavior of the student program. For example, a test program can implement fuzzy comparison of floating-point values when grading programs where the numerical accuracy is not strictly specified, thus allowing a range of student solutions to be accepted as correct. Test programs can also measure and evaluate the performance characteristics of the student program for performance testing. Figure 7 shows a block diagram of the output analysis strategy.

3.4 Stream Control

Stream Control is a strategy for console programs that interleave input and output streams. The test output for such programs combines program output, including prompts, and program input. Ideally, these should appear in the same order that the user would see if they ran the program interactively. This is problematic in the batch environment where input data is redirected from a file to simulate the user typing, since without special handling the input stream will not be echoed to the output stream. Many grading frameworks and online tutorials avoid this problem by displaying separate windows for input and output.

Figure 8 shows test results where the input and output have been ordered correctly using stream control. The assignment from which these results were generated prompts for income data and reads the data with a Java scanner, performs a tax calculation, and outputs results. A capitalization error has caused a failure, which is color coded for easy recognition, and the user input for this example is redirected from a file. By interleaving prompts, input, and output, the feedback to the student is more like what they would see when running the program.

Our solution for stream control is implemented in a test program called Runner that handles a variety console-based programs, regardless of the number of inputs or content of the prompts. Runner executes the student program within a child process, then spins up threads to process the program's input and output. The output handler continually reads and prints output from a buffered reader attached to the output

stream of the student program. The input handler intercepts the input stream and detects when the program is waiting for input. It then feeds the inputs to the program and echoes them to the output stream. Figure 9 shows a block diagram of the stream control strategy.

The primary technical challenge for stream control is to reliably detect when a program or process is waiting on input. We have now developed two solutions, both of which are dependent on the Linux platform. The first is a Perl script that queries the state of the Linux process, and the second uses Java code to snoop the stack trace of the student program during execution. The latter is embedded in the source code for Runner. Runner also has a timeout thread to terminate student programs that enter infinite loops or expect the wrong number of inputs, based on an amount of time specified by the instructor. The termination problem is discussed in more detail in Section 4.

3.5 Object Instantiation

Object Instantiation is implemented through test programs that directly instantiate student classes in order to verify their implementation through method calls and member accesses. This approach models unit testing, which we believe is an important concept for students. Instantiation can additionally support module testing by verifying the interactions between methods and state. To implement an instantiation test, the instructor must customize test code for each method call or member access.

The main method in the student code is not usually tested via instantiation, however this is technically possible. Some instructors require the students to write test code for their class in main, and such test code is easily verified within our framework. A limitation of the strategy is that students must specify all of the methods with correct parameters (in order), return types, and attributes, otherwise the compile will fail and the student will receive a zero on the assignment. If the program does compile, then detailed feedback can be provided on the functionality of the student program.

A workaround for the compile problem is to require students to implement an interface in their classes. This forces students to correctly specify methods to allow their programs to compile. Another solution is to use Java reflection, which is discussed as a separate strategy. Figure 10 shows a block diagram of the object instantiation strategy.

3.6 Instrumentation

Instrumentation is a strategy that is useful when classes are supplied to the student as part of the assignment. The basic idea is to ignore the output of the student program, and embed test code into the supplied classes instead. For example, we often introduce graphical user interface (GUI) into assignments in our introductory courses. Students at this level are not usually advanced enough to program GUI

```
TEST NAME: test1 COMMAND LINE: java Runner Program2 5 < test2.input

Gross salary? 73685.73                 Gross Salary? 73685.73
Interest Income? 1087.89               Interest Income? 1087.89
Capital Gains? 12682.50                Capital Gains? 12682.50
Total Income: $87456.12                Total Income: $87456.12
Adjusted Income: $78456.12             Adjusted Income: $78456.12
Total Tax: $7815.67                    Total Tax: $7815.67

test1 Score: 0 / 10
```

Figure 8: Test Results from Stream Control.

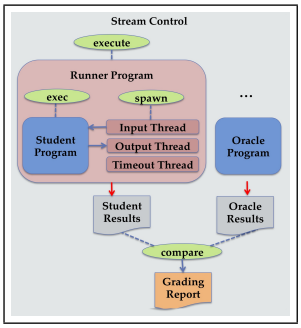

Figure 9: Stream Control Strategy.

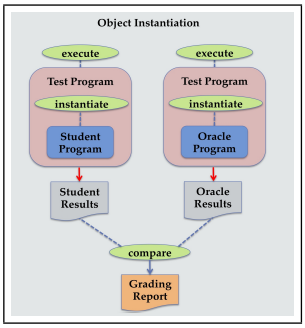

Figure 10: Object Instantiation Strategy.

applications, but students find assignments that incorporate user interface to be compelling.

When using instrumentation, testing proceeds through instrumentation of the supplied classes that captures how and when methods are called. The test script discards standard output, thereby ignoring output from the student code. Test output from the supplied classes is sent to standard error, which is used exclusively for comparison against the oracle. In addition, we can remove GUI calls from the supplied class to make testing faster and more efficient. The benefit of instrumentation is targeted and detailed feedback on the behavior of the student program. Figure 11 shows a block diagram of the instrumentation strategy.

To further understand instrumentation testing we consider two assignments. The first is a maze program in which the student code instantiates a supplied GUI object and writes code to control the behavior of a character in a maze. The second is a plotting program in which the student code reads data from a file and uses a GUI object to display color charts and graphs. For both assignments the student must

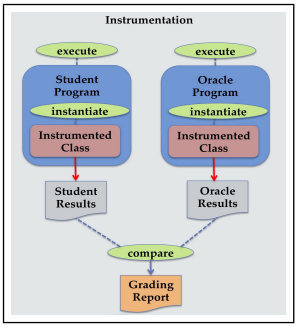

Figure 11: Instrumentation Strategy.

instantiate the supplied GUI class and invoke methods on it. The structure of the student code is not specified for these assignments, so there is no way to directly test student code. Instead we verify the student program indirectly by checking method calls to the supplied object to make sure the student code has implemented the correct behavior.

For the maze program, the supplied code simply prints out the direction each time the student code calls our object to move the character. If the student correctly follows the specified algorithm, then their moves will be identical those of the oracle program. Incorrect, superfluous, or duplicate moves will cause test failures. For the plotting program, we check the size and contents of the data arrays by printing them. If student code fails to read the file correctly, or fails to allocate arrays dynamically, then test failures will occur. Supplied code can check other behaviors such as the order and number of method calls.

3.7 Reflection

Programming languages such as Java and Python provide reflection, which enables introspection of class members and methods. This feature has proven to be highly useful in writing tests. Reflection code can enumerate members and methods in student classes, and invoke methods without instantiation. Enumeration includes data types, access modifiers, static keyword, return values, and generic (nameless) declarations of method parameters. This information provides the student with feedback on how closely their class matches the specification.

The ability to invoke methods without instantiating is a key advantage. Reflection also allows private members and methods to be examined and invoked. Thus partial credit can be awarded for methods that are implemented correctly, without the compile errors that can result from instantiation. This strategy therefore provides more precise feedback than instantiation testing. Figure 12 shows a test result where methods in a student class are enumerated. The student has failed the test because a method was declared as private instead of public. Figure 13 shows the reflection code that enumerates the methods in a class.

Reflection tests are easy to leverage from previous assignments, since the code to enumerate methods and member variables is identical from one assignment to the next. However, method invocations and functional tests have to be tailored to the assignment. This strategy is useful for languages that support reflection, as well as programming languages such as C++ that have third party solutions for reflection. Figure 14 shows a block diagram of the reflection strategy.

```
TEST: test1 CMD LINE: java TestProgram3

Your Output/Master Output:

  Verifying methods in Program3.java:
  public static int Program3.sumArray(int[])
  public static char[] Program3.createChars(java.lang.String)
  private static double Program3.findLargest(double[])
  public static int[] Program3.translateChars(char[])

  Verifying methods in Program3.java:
  public static int Program3.sumArray(int[])
  public static char[] Program3.createChars(java.lang.String)
  public static double Program3.findLargest(double[])
  public static int[] Program3.translateChars(char[])

test1 Score: 0 / 10
```

Figure 12: Test Results from Reflection.

```
// Java reflection code to enumerate methods in a class
public static boolean queryMethod(String className) {

    // Reflection code to enumerate methods in a class
    try {

        // Search for class
        Class<?> c = Class.forName(className);

        // Enumerate methods
        Method[] allMethods = c.getDeclaredMethods();
        for (Method m : allMethods) {

            // Get method declaration
            String methodDeclaration = m.toGenericString();

            // Print method declaration
            System.err.println(methodDeclaration);
        }

    } catch (ClassNotFoundException x) {
        System.err.println("Could not find: " + cName);
    } catch (InstantiationException x) {
        System.err.println("Could not instantiate: " + cName);
    } catch (IllegalAccessException x) {
        System.err.println("Could not access: " + cName);
    }
}
```

Figure 13: Listing of Reflection Code.

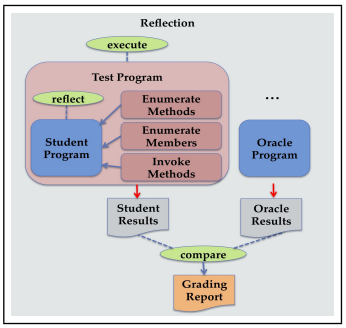

Figure 14: Reflection Strategy.

3.8 Code Analysis

With a flexible framework, the instructor is not limited to functional testing as described above. Test programs can also read and parse student source code, thus opening the door for various kinds of analysis. Code analysis can be implemented as a custom test program, or in some cases by using other tools such as static analysis programs.

To date we have made limited use of code analysis by verifying that the program header is correctly formatted, that sufficient comments are provided, that required methods are called, and that proscribed methods are not. However, we can easily envision grading based on program style, code complexity, code coverage, and so on. Figure 15 shows a block diagram of the code analysis strategy.

4. ISSUES

Regardless of the implementation of an automated framework, several issues consistently emerge. The first is that programs submitted by students often fail to terminate, thus requiring a robust timeout mechanism in the framework. The second is security, which must be addressed to protect against inadvertent and intentional attempts to compromise the testing system. The third is the performance required to make a grading system interactive.

4.1 Non-Terminating Programs

Student programs often exhibit non-terminating behaviors that interfere with automated grading. We commonly encounter programs that have infinite loops (with no input or output), that produce infinite output, or that expect the wrong number or type of inputs. The latter may hang waiting for unspecified input that is never provided. The prevalence of such programs has led us to incorporate timeout code into the grading framework, as have other institutions [3]. One such mechanism is provided by the Runner program, which has previously been presented. In addition to stream control, Runner differentiates between infinite loops and student programs that expect too much or too little input, and can truncate excessive output.

4.2 Security Issues

Testing frameworks must consider the possibility that code submitted by a student could compromise the test system. These threats include (but are not limited to) unauthorized access to class materials, theft of submissions from other students, access to or modification of grades, and damage to the integrity of the test system. Students in our introductory courses have not proven to be sophisticated enough to intentionally attack the grading system, but we have had at least one submission that inadvertently caused a problem by overwriting input files.

We have identified and implemented a number of strategies to maintain security in our framework. The first line of our defense is authentication, in which our framework verifies that a student is registered in the course and holds valid credentials for department systems. Anyone that does not pass the authentication is unable to submit to or interact with the test system. Beyond authentication, we increase security by isolating testing from class directories to prevent access to exam and assignment solutions and other sensitive materials. One way to do this is to have the framework set file protections to prevent student code from reading or writing files that they should not have access to. A variant of this technique is to run student code under a different Linux user or group to prevent access.

We are investigating solutions that provide even more protected environments. One proposal is to perform testing using the Linux *chroot* command. Another proposal is to test student programs in the context of a virtual machine. Either method precludes access to class directories and other materials, and helps protect the integrity of the test system. A more elaborate security system could detect and log student submissions that attempt unauthorized access or other damaging actions. Commercial system for automated grading will undoubtedly have to address these concerns in the future in order to sell into academic markets.

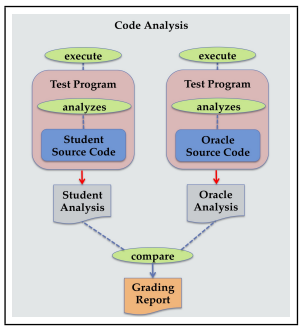

Figure 15: Code Analysis Strategy.

4.3 Grading Performance

To support interactive testing, a grading system should be able to test a student submission in a small number of seconds. The performance of our framework is less than a second per test case per student, not counting the duration of the student program. Since programs in introductory courses usually execute quickly, we can support interactive testing with only a single-threaded server running on a single system. In batch mode, classes with several hundred students can usually be graded in under an hour, without supervision. This represents a large time savings compared to manual grading. Future plans include a test server that is multi-threaded to decrease latency.

5. CONCLUSIONS

This paper has described a set of strategies for testing student programs that are more effective than regression testing at providing detailed and relevant feedback to students. In addition, we have discussed some of the issues that arise in the context of automated grading and their solutions.

6. ACKNOWLEDGMENTS

The author would like to acknowledge Fritz Sieker for writing the automated grading framework, and Jack Applin for writing the initial version of the Runner program.

7. REFERENCES

[1] C. Douce, D. Livingstone, and J. Orwell. Automatic Test-based Assessment of Programming. *Journal Educational Resources Computing*, 5(3), Sept. 2005.

[2] S. Edwards. Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In *Proc. International Conf. on Education and Information Systems: Technologies and Applications (EISTA 03)*, pages 421–426, 2003.

[3] S. H. Edwards, Z. Shams, and C. Estep. Adaptively identifying non-terminating code when testing student programs. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 15–20, New York, NY, USA, 2014. ACM.

[4] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu. APOGEE: Automated Project Grading and Instant Feedback System for Web Based Computing. In *Proc. of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 77–81, New York, NY, USA, 2008. ACM.

[5] M. J. Hull, D. Powell, and E. Klein. Infandango: Automated Grading for Student Programming. In *Proc. of the 16th Annual Joint Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 330–330, New York, NY, USA, 2011. ACM.

[6] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proc. of the 11th Annual SIGCSE Conf. on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 13–17, New York, NY, USA, 2006. ACM.

[7] C. Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, New York, NY, USA, 2015. ACM.