Spring 2016

Semi Automated Partial Credit Grading of Programming Assignments

Thomas Richard Rossi
University of New Hampshire, Durham

SEMI AUTOMATED PARTIAL CREDIT GRADING OF PROGRAMMING ASSIGNMENTS

BY

THOMAS R. ROSSI
BA, Washington and Jefferson College, 2008

THESIS

Submitted to the University of New Hampshire
In Partial Fulfillment of
The Requirements for the Degree of

Master of Science
In
Computer Science

May 2016

SEMI AUTOMATED PARTIAL CREDIT GRADING OF STUDENT PROGRAMS

BY

THOMAS R. ROSSI

This thesis has been examined and approved.

Thesis Director, R. Daniel Bergeron, Professor, Computer Science

Elizabeth Varki, Associate Professor, Computer Science

Mark Bochert, Lecturer, Computer Science

Arvind Narayan, Lecturer, Computer Science

On April 20th, 2016

Original approval signatures are on file with the University of New Hampshire Graduate School.

DEDICATION

Firstly, I would like to thank Dr. R. Daniel Bergeron for being willing to serve as my thesis advisor. Without a doubt, this work would not be where it is without him taking my ideas and helping them become something bigger. Through this process he has helped me understand what it takes to deliver quality work in academia, and I cannot thank him enough for that. He has also been an amazing mentor to me during my entire time at UNH and I have learned so much from him.

I would also like to thank my mother and stepfather for having been there to support me through this entire experience as much as they did. They believed in me even when I was unsure of myself. I do not know what I would do without them. While not with me anymore, I also know that my father was cheering me on in his own way. Whenever things were down, I somehow knew exactly what he would say to keep me going, to keep me on the right path.

Of course, I would be remiss if I did not thank all of the faculty in the Computer Science Department who helped me along the way. Amongst them, I would like to specifically thank Dr. Elizabeth Varki, Dr. Mark Bochert, and Arvind Narayan for everything they taught me over the years and their willingness to then serve on my thesis committee. They have taught me so much about this discipline and also helped me become a great teaching assistant to the students I was fortunate enough to mentor during my time in the department.

I am truly blessed to have each and every one of you in my life, thank you very much for everything you have done for me throughout this process. This work would not exist without each of your unique contributions and I cannot thank you enough for that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

SEMI AUTOMATED PARTIAL CREDIT GRADING OF PROGRAMMING ASSIGNMENTS

by

Thomas R. Rossi

University of New Hampshire, May 2016

The grading of student programs is a time consuming process. As class sizes continue to grow, especially in entry level courses, manually grading student programs has become an even more daunting challenge. Increasing the difficulty of grading is the needs of graphical and interactive programs such as those used as part of the UNH Computer Science curriculum (and various textbooks).

There are existing tools that support the grading of introductory programming assignments (TAME and Web-CAT). There are also frameworks that can be used to test student code (JUnit, Tester, and TestNG). While these programs and frameworks are helpful, they have little or no no support for programs that use real data structures or that have interactive or graphical features. In addition, the automated tests in all these tools provide only "all or nothing" evaluation. This is a significant limitation in many circumstances. Moreover, there is little or no support for dynamic alteration of grading criteria, which means that refactoring of test classes after deployment is not easily done.

Our goal is to create a framework that can address these weaknesses. This framework needs to:

1. Support assignments that have interactive and graphical components.
2. Handle data structures in student programs such as lists, stacks, trees, and hash tables.

3. Be able to assign partial credit automatically when the instructor can predict errors in advance.
4. Provide additional answer clustering information to help graders identify and assign consistent partial credit for incorrect output that was not predefined.

Most importantly, these tools, collectively called RPM (short for Rapid Program Management), should interface effectively with our current grading support framework without requiring large amounts of rewriting or refactoring of test code.

CHAPTER 1

INTRODUCTION

1.1 Programming Assignment Grading

The introductory courses in Computer Science at UNH are heavily focused on interactive graphics-oriented programs. These programs are tedious to grade as they have numerous components that must be evaluated by running each program interactively. Making this problem worse is the fact that enrollments in the entry level Computer Science courses are on the rise.

There are existing tools that support the grading of introductory programming assignments (TAME [1] and Web-CAT [2]) as well as various testing frameworks that can potentially be used (JUnit [3], Tester [4], and TestNG [5]). While these programs and frameworks are helpful, they have several shortcomings:

- They have no effective support for partial credit evaluation which means that all automated tests are "all or nothing" in terms of credit.
- They support use of only the most basic data structures.
- There is little or no support for dynamic alteration of grading criteria, which means that refactoring of test classes after deployment is not easily done.
- There is no support for interaction and graphics.

The goal of this research is to develop a framework, RPM (short for Rapid Program Management), that facilitates program grading with a particular focus on the challenges of grading more complex programs that use list and tree data structures and interactive graphics, and for which

it is important to be able to give partial credit. We propose a new approach in which the grading is done in a semi-automated fashion. The program should do most of the grading, but involve a human grader for ambiguous results or when human interaction with the program cannot be simulated. The framework should be able to provide information that helps the human grader more rapidly assess the student work, facilitate partial credit grading, and provide the ability to modify a test set after deployment.

There needs to be a way of describing the tests that will be run against student code. These tests can optionally have *subtests*, subcomponents that allow for more modular tests against programs involving complex entities (e.g., tree data structures). Each test must result in an *answer* in some form that can be compared to an answer predicted by the instructor, usually produced by a sample solution. For each possible test answer, the instructor needs to specify the percentage of credit to give for that answer. It is also necessary to make a connection to specific rubrics in TAME that a test affects. This makes it possible to migrate the data over to TAME once all processing has been completed. Tests and subtests can also compute and save numerical metadata that might help to classify a complex answer such as a list or tree. In doing so, clusters of similar wrong answers can be formed that allow the grader to have more insight when attempting to establish partial credit for a specific rubric in an assignment.

1.2 Assignment Test Set

Every assignment needs to be tested by instructor written code designed to test the specific features unique to that assignment. For a very simple program, like a program to compute the average price of stock shares in a linked list, there is only one correct answer so the test is very easy to write. When the program is more advanced, such as a program to compute the path to a specific node in a binary search tree, testing becomes a little more complicated since there are

more places where students can make mistakes, but arrive at solutions that warrant some amount of credit. At UNH, first-year students are required to implement significantly more complicated programs such as the correct playing of a solitaire card game. Although the evaluation of the graphical and interactive components of such assignments are very difficult to automate, appropriate assignment specification can lead to feasible approaches to test the correct implementation of the game play. This requires playing the game multiple times with different card ordering and comparing the student's end game state with that of the sample solution as well as a rigorous specification for the output format for the game state that is usually written as an output file. Comparing the student answer with the correct answer requires tools for comparing text files representing the final game state.

The test set also includes annotations by a preprocessor that provides information to the test support library for use at test runtime. Each unit test in the test set will call the test support library methods for each test for each student to report the results.

1.3 Commentary Language

Javadoc-like annotations in the testing code describe all of the necessary aspects of each test including expected potential answers, both correct and incorrect, and the percentage of points each answer is worth. The answers can be specified as numerical values, exact strings, regular expressions, or files. The information in the commentary annotations is stored in a database that is used while testing student submissions.

1.3.1 Defining Tests and Subtests

Defining a test in the commentary language allows the test code to be linked to rubrics, answers, and metrics. A test is a single set of inputs and conditions intended to evaluate a feature

of the student's code. Subtests allow for one test to be mapped to different sets of rubrics and answers for each input and set of conditions used.

1.3.2 Connecting to TAME Rubrics

A single test or subtest can be connected to any number of TAME rubrics with a specified contribution to that rubric. Once all grading has been completed, the data can be aggregated for each rubric and easily migrated over to the appropriate TAME database tables. When the grader enters the TAME environment, the results of the automated grading are available so they can focus on only those rubrics that were not fully graded.

1.3.3 TestLoader

The TestLoader program reads a Java file, extracts the commentary language information, and either outputs it as a CSV file or stores it directly into a database for later use in the pipeline. The determination as to whether the output is sent to a CSV or to a database is made at run time. The TestLoader is designed so that it can be run as a batch style program and can therefore be integrated into other pipelines as needed.

1.4 Test Support Library

Test support library functions are invoked during test set execution to save and analyze the results of student program tests. It correlates a student's test answer with an answer provided in the commentary language and supports the recording of metadata metrics so they can be used later to analyze student results if needed.

1.4.1 Handling Students with Predicted Answers

When a student's results are passed to the test support library, the results are compared to the list of predefined answers for the test. If a match is found, the associated value of that answer is recorded alongside the student's answer and sent either to a CSV file or database.

1.4.2 Handling Students with Unpredicted Answers

While many "wrong" student answers can be predicted based on known patterns of behavior for a certain assignment, some students create unexpected answers. In this case, we want to provide additional information to the human grader that might facilitate grading programs for these students. For example, we can group all the students with the same wrong answer into a cluster that can be graded at the same time. This allows the grader to determine an appropriate partial credit and award it to all the students in the cluster. The ability to group students with identical wrong answers leads naturally to the desire to group students with similar wrong answers and to the concept of creating metrics that can be applied to student answers for the purpose of clustering wrong answers.

1.4.3 Metric Data

Metrics connect to individual rubrics for a specific test and are used to help understand the student's answer. Metric values can represent anything that could be used to help the grader determine partial credit for a specific rubric for a certain test. Metric data can be captured not just for complex data structures, but also for simple answers such as numerical or string answers by calculating the distance between the student's answer and the correct answer.

In the case of advanced assignments, such as those using unique data structures more complicated than an array or simple list, it is very hard to measure the partial correctness of a solution based solely on a textual comparison of the "answer". It is possible, however, to calculate some metrics associated with each student's data structure and compare the student's metric values to those of a sample solution and to help cluster similar wrong answers in order to facilitate the determination of appropriate and consistent partial credit. Examples of data structure metrics can include the number of nodes in the data structure, the number and percentage of values in the

student solution that do and do not appear in the sample, etc. The system provides a small library of basic metrics for common data structures, but the instructor is able to define arbitrary additional metrics that are appropriate for specific assignments and rubrics.

1.4.4 Clustering

It is easy to create clusters of identical unexpected answers. These clusters, while valid, usually produce mostly singleton clusters, which is not very helpful. Instead, we would like to create clusters of similar answers. By using the metric data collected for unpredicted student answers, clusters can be comprised of students whose answers are not the same, but the metric data on their answers makes them similar. There can be multiple clustering algorithms implemented to create these groupings.

Clustering is used in the pipeline to bring unpredicted similar results together into meaningful groupings. In doing so, it is possible to handle all the unpredicted answers that fit a certain profile together, which should reduce grading time and increase grading consistency.

1.5 GradingKit

GradingKit provides a straightforward object-oriented interface to the information in the database. Designed using the model, view, controller (MVC) pattern, GradingKit contains both controllers and models for each of the tables in the database. Models represent records in each of the tables in the database; controllers abstract away the JDBC interactions and result set management that has to take place. Their input parameters identify a specific query to be made to the database and they return the model objects that satisfy the query.

1.6 Abbot

Abbot [6], an open source library for testing graphics, provides a mechanism to automate interactions with graphical components (shapes, UI elements, etc.). Since the interactions are

programmed by the user, they can be repeated every time the program is run. JUnitE integrates with Abbot and can take in information Abbot generates as the result of a specific test on the student code.

CHAPTER 2

BACKGROUND

Existing tools for automated grading of student programs have typically been developed within the context of either general-purpose program testing or program management tools.

2.1 Program Testing

There are a number of existing testing frameworks that can support the grading of student programs including JUnit [3], Tester [4], and TestNG [5].

2.1.1 JUnit

JUnit is an industry standard framework that is used by many companies to test software written in Java and most Java IDEs support JUnit testing. This framework tests assertions made by the user. For example, a user may test that a variable is not null or that two values or objects are equal, or within some tolerance of each other. A full list of JUnit tests can be found in the Java Assertion class API. Tests are either pass or fail; if the test fails on one of the assertion methods, an Assertion Error is thrown. Since IDEs have certain modes for running JUnit tests versus programs, they typically catch and handle these exceptions so the test class does not have execution halted. This framework is beneficial since it can catch exceptions thrown by student code (such as a null pointer). Figure 1 shows an example JUnit test for an assignment to read a set of numbers from an input file into a linked list. This test verifies that a student method to compute the average of a linked list will compute the correct answer, 1.5, based on a linked list generated by a static method in the `App` class.

```
@Test
public void testAverage()
{
```

```
        LinkedList<Double> list = App.makeList("numbers.txt");
        double avg = App.computeAverage(list);
        assertTrue(avg == 1.5);
}
```

Figure 1: Example of JUnit Code

2.1.2 Tester

Tester provides a simpler testing environment than JUnit in order to make testing more accessible to students in introductory programming courses [7]. The Tester library [4] uses reflection to perform analysis on source code. A student's program can create an object representing the result of a specific assignment task then ask Tester to compare it to an object representing the correct answer. If a test fails, the contents of both two objects are output for user analysis. There are also methods to compare values of primitive Java types (floating point numbers, integers, etc.). Figure 2 shows a test using Tester to validate that a student's `reduce()` method correctly changes the values in the `Item` class.

```
Item bread = new Item("Bread", 100);
Item milk = new Item("Milk", 200);

Void testReduce(Tester t)
{
      t.checkExpect(bread.reduce(20), new Item("Bread", 80));
      t.checkExpect(milk.reduce(30), new Item("Milk", 170));
}
```

Figure 2: Example test using Tester framework [4]

2.1.3 TestNG

TestNG (short for Test Next Generation) [5] is a framework designed to test not just individual Java classes, but the entire range of testing goals from unit to functional to integration tests. The developers of TestNG based their design on how simple JUnit is to use and how programmers can use annotations to handle configuration tasks [8]. TestNG contains various metadata annotations that allow the programmer to indicate what should happen before or after the

execution of suites, tests, classes, or methods. For example, `expectedExceptions` allows the programmer to indicate a list of possible exceptions that the test may throw. If any of these exceptions are generated, TestNG marks the test as a pass, if an exception not on the list is thrown, the test is marked as a fail [5]. In Figure 3, a sample TestNG test is shown. The test program intentionally overbooks the plane. The annotation after `@Test` indicates this test is expected to throw an exception of type `ReservationException`. Should another type of exception be thrown, the test fails.

```
@Test(expectedExceptions = ReservationException.class)
public void shouldThrowIfPlaneIsFull()
{
      plane plane = createPlane();
      plane.bookAllSeats();
      plane.bookPlane(createValidItinerary(), null);
}
```

Figure 3: Example test using TestNG [5]

2.1.4 Abbot

Frameworks such as JUnit, Tester, and TestNG do not have a way of interacting with GUI elements in a program. Abbot [6] is designed to fix this problem and provide a mechanism to test GUI elements in an automated fashion similar to what one might find in JUnit or Tester for non-GUI type code.

Abbot is not a testing framework in and of itself, rather it is something a programmer would use in conjunction with something like JUnit or TestNG. While the programmer is free to hand code graphical interactions in a test, it is also possible to record user interactions as a script through Abbot's companion program, Costello [6]. The Abbot framework still tests assertions similar to what JUnit does. It is the result of these assertions that determines test success or failure. Figure 4 shows a sample test method using Abbot to test the functionality of a start button in a student program. The test calls Abbot to find two buttons on the screen, with labels "Start" and "Stop" using the `getFinder()` method. It then simulates a click of the stop button followed by a click

of the start button by calling the `actionClick()` method of the `JTextComponentTester` class.

```
public void testStartButton() throws Throwable
{
   MoverApp app = new MoverApp( "Mover App Demo", new String[ 0 ] );

    JButton starter = (JButton)getFinder().find(new Matcher() {
       public boolean matches(Component c) {
          return c instanceof JButton &&
             ((JButton)c).getText().equals("Start");
       }
    });

    JButton stopper = (JButton)getFinder().find(new Matcher() {
       public boolean matches(Component c) {
          return c instanceof JButton &&
             ((JButton)c).getText().equals("Stop");
       }
    });

    JTextComponentTester tester = new JTextComponentTester();
    tester.actionClick( stopper );
    tester.actionClick( starter );

    assertTrue( "Should be true", app.isRunning() );
}
```

Figure 4: Sample Abbot test code

2.2 Program Management

Program Management tools serve as vehicles that facilitate the grading of student work. While these tools provide everything from submission to returning mechanisms, our interest here is limited to the support for grading. The sections below examine the capabilities of the TAME program management systems used at UNH as well as Web-CAT, a popular program management tool for test-driven development curriculums.

2.2.1 TAME

In Fall 2013, we significantly improved the program grading process at UNH with the first version of TAME (Tool for Assignment Management and Evaluation). Although TAME provides only minimal automatic grading, it does provide a nice graphical environment in which the grader is presented with the rubrics for the assignment, the student's output, and the sample solution output side-by-side with non-matching lines highlighted. When the two outputs match for a particular pre-specified rubric, the grader only needs a one button click to enter the points into the grading output form. If the output is completely wrong, only one click is needed to deduct all the points for that rubric. This feature greatly simplifies the grader's task for many labs and for some parts of some programming assignments. TAME also calculates deductions for late submissions and for style and automatically factors those into the final grade. As the assignment is graded, TAME updates the student's final score based on the values entered for the various rubrics.

2.2.2 TAME Grading Assistance

TAME assists the grader by providing a graphical interface for grading and maintaining grading results, generating grade reports, and overall management of the grading process. This interface allows the grader to ensure that all of the necessary items are evaluated and identify points that need to be deducted. The output comparison tool is an efficient and effective mechanism for comparing batch output. Using this, the grader is able to focus on the differences between the two outputs and assume that any outputs that do not have any lines highlighted are successful tests.

The software also provides some automation. It first looks to see if the student's submission date warrants the deduction of any points based on the submission date and allowed late days associated with the assignment. The grading system can also run a style check on the student's code to ensure that the proper style guidelines were followed and if need be, deduct

points for poor style as specified in the assignment description. Any points deducted by either of these two automated steps are factored into the student grade along with the deductions the grader makes to compute the student's final score.

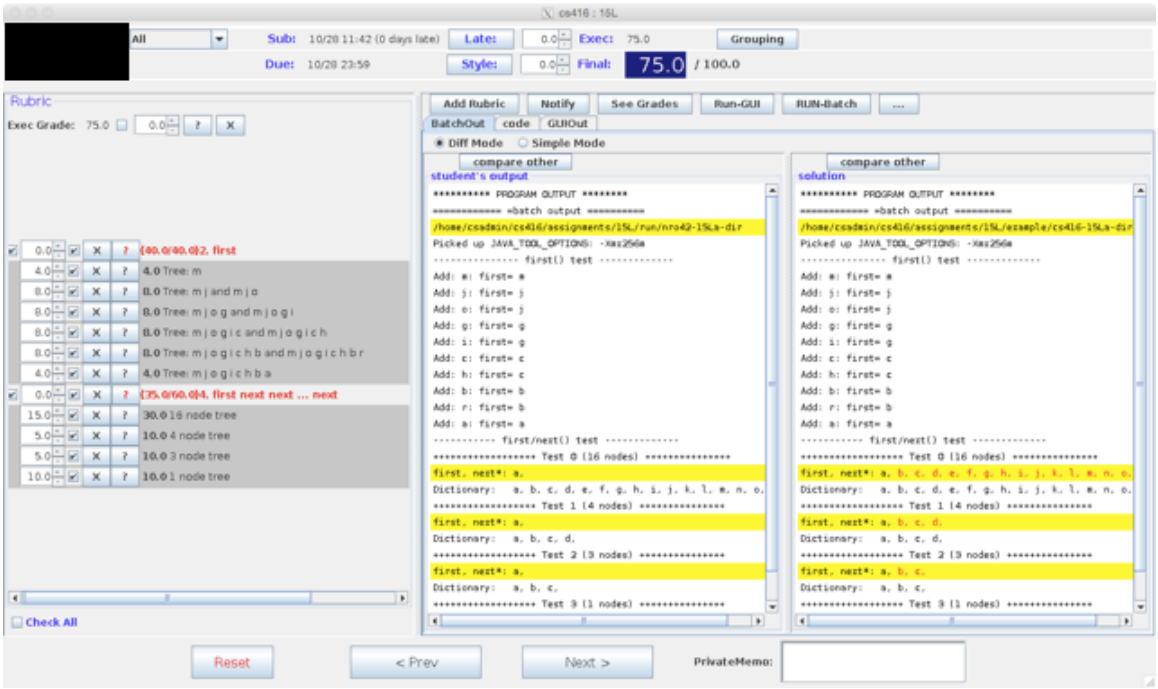

Figure 5: Screenshot from TAME

2.2.3 Web-CAT

Web-CAT is designed to assess student work in classes using test driven development curriculums. The student's code is first assessed to see how well it can pass the tests the student has written, this is referred to as "code correctness". The intent is to build a student's self confidence in their own ability to test. Secondly, a student's tests are examined for code coverage, referred to as the "test completeness". This is done by using instrumentation to determine what sections of code the student's tests ran and which did not. The granularity of this can be controlled by the instructor and can include checks at various levels of student code (methods, conditionals, etc.) Finally, Web-CAT assesses what it calls "test-validity". Here, students are assessed as to how valid their tests are. This is done by comparing how well a student's tests match the problem

they are being asked to solve. While the plugin for Java does assess student work, it is also designed to evaluate a student's style and ensure that it matches coding conventions. Plugins do exist for other languages which means Web-CAT is not just restricted to Java based curriculums. Web-CAT goes also handles student submissions, supports code mark-up by graders, and returns both the final grade and the mark-up for the program to the student [2]. It also has plugins so student work can be submitted from and graded in Eclipse. According to the online documentation, developers can write their own plugins, but documentation does not seem to be available.

Although Web-CAT is a powerful tool, it has two major shortcomings: 1) it is not able to award partial credit for a particular test and 2) it is not easy to handle complex data structures (lists, trees, etc.).

CHAPTER 3

FUNCTIONAL OVERVIEW

The RPM grading support framework is both powerful and flexible. It is powerful in the sense that it provides a collection of tools that reduce the difficulty of grading student work in an efficient manner. It is flexible because that the software does not impose any one specific instructor or institution's way of grading. Moreover, the software is designed to be extensible in the future so that additional features can be added to the platform.

3.1 Writing the Test Set

The instructor uses RPM's JUnitE to create a test set that contains all the tests to be run against the student code. The instructor must also define any code that needs to run as part of the setup and teardown for the test set and for each individual test, such as opening and closing files and creating an instance of a data structure object to be passed to the student's code.

3.2 Commentary Language

The commentary language provides the necessary test information to the TestLoader program and provides structure to that information. It has the look and feel of Java's "`@`" comment tags and is interspersed in the test set code provided by the instructor. It is used to describe multiple possible answers for a test, the rubrics affected by the test, the weight of that test for each rubric, and other information useful in the grading process. All arguments associated with the tags are tab delimited.

3.2.1 Creating Tests

Test tags are the key to defining the relationships between tests, rubrics, answers, metrics, etc. The `@TestName` tag, defines the start of a test specification, which ends with an `@endTest` commentary language tag:

```
/**
 * @TestName <name>
 * @TestDesc <description>
 * .
   <Other commentary language tags as appropriate>
   .
 */
<code>
/** @endTest */
```

A sample tag is shown below:

```
/**
 * @TestName Sorted
 * @TestDesc One-way list test
 * .
   .
   .
 */
<JUnitE Test>
/** @endTest */
```

In this example, a new test is defined with the name “Sorted” and the description, “One-way list test”. This information, combined with additional tags present in the commentary language distill necessary knowledge about that test into a form that can be parsed and stored in the database.

3.2.2 Writing Subtests

A single test can be composed of multiple components, or subtests. This can be used, for example, to test a search method of a data structure with multiple inputs, perhaps one for each element that should be in the list and some that should not.

In the case of subtests, the `@TestName` tag defines the prefix for the subtest name followed by an incrementing integer value starting at zero. For example, if there are five subtests for test "A", then they would have the ids A0-A4.

```
/**
 * @TestName     TREE
 * @TestDesc     Binary search tree tests
 * .
 * <Other commentary language tags as appropriate>
 * .
 * @SubTest
 * @Rubric  1    1    1.0
 * @Com     Path to Brian:
 * @Ans     j.k  1.0
 * @Com     Path to Brian: Extra . in path
 * @Ans     j.k. 0.95
 * @SubTest
 * @Rubric  1    2    1.0
 * @Com     Path to Ken:
 * @Ans     j.g.f.b   1.0
 * @Com     Path to Ken: Extra . in path
 * @Ans     j.g.f.b.  0.95
 * .
 * <Other commentary language tags as appropriate>
 * .
 */
```

Figure 6: Example of a subtest

The test associated with Figure 6 invokes the student binary search tree code to search the tree for a specific node. The student's code should return a string that represents the path to the node. This string is sent to the test support library that either maps the student result to one of the predefined answers for that subtest, or flags it for review by a human grader.

3.2.3 Connecting to Rubrics

The `@Rubric` tag connects a test (or subtest) to a specific TAME rubric specification using the following syntax:

```
@Rubric <rubric-section> <rubric-number> <weight>
```

For example,

```
@Rubric    3     1     0.5
```

specifies that this test (or subtest) represents 50% of the points associated with the TAME rubric item 1 in rubric section 3.

Rubrics can be specified on a per test or per subtest basis and any number of rubrics can be associated with a test or subtest. In the case where subtests are used, each of the subtests can be connected to different rubrics. Similarly, the weights of the connections in one test or subtest have no bearing on the weights of the connections in another test.

3.2.4 Defining Answers

The `@Ans` tag, defines a potential answer and the percentage of the points that answer is worth. The matching of a student answer to a predefined answer is determined by a variety of user-specified parameters including, for example, string comparisons that ignore case and/or space. The format of the `@Ans` tag is:

```
@Ans<:matching rules, if any> <answer to match> <point ratio>
```

For example:

```
@Ans 59.75      1.0
```

defines a new answer for a test or subtest where the answer is 59.75 and the ratio of points awarded for that answer is 100% of the possible points. An answer can be a Java string, a numerical value, or a file specification defining a file that contains the answer.

The `@RegAns` tag defines a regular expression that indicates a variety of possible potential answers and their value. Similar to the `@Ans` tag, the match stringency can be adjusted according to the various flags that are available in Java for regex pattern matching. The format for the `@RegAns` tag is:

```
@RegAns<:matching rules, if any> <regex to match> <point ratio>
```

For example:

```
@RegAns   a*b* 1.0
```

defines a new regular expression answer that matches to any number of a's followed by any number of b's and is worth 100% of the possible points.

The `@Com` tag which appears before an answer (`@Ans` or `@RegAns`) is used to define a comment the student will see regarding the answer they match against and also accepts a comma delimited set of keywords that are associated with the answer for later analysis. The format of this tag is:

```
@Com <comment for student> [<comma delimited keywords>]
```

The example tag shown below (and later ones) use ↦ to show a tab for clarity:

```
@Com ↦ You are missing a node
```

or:

```
@Com ↦ You are missing a node ↦ missing node
```

In both cases, the student is told they did not include all of the nodes. The keywords can be used by system utilities to gather information about grading for clustering or analysis on a subtest basis

3.3 TestLoader

The TestLoader is the first program run as part of the software pipeline; it converts the commentary language information into either a CSV file or a SQLite database that has predefined database tables already installed.

3.3.1 Parsing Commentary Language Tags

When the TestLoader encounters a line beginning with a comment tag, the entire comment is read and broken into a set of tags and values. The data for each tag is read and stored. The code for the test is also captured and stored. If the comments do not properly conform to the logic rules

for parsing, the parse of the test is considered to be a failure. If the parse is successful, the data is sent to an SQLite database or written as a CSV file.

3.3.2 Database Output

An output filename with the extension of "`.db`" triggers the TestLoader to output all parsed information into a predefined SQLite database that has been pre-populated with certain tables. This mode supports the test/rubric correlation and information defining clusters for students with similar answers that do not match any predefined answers.

The data obtained from the pipeline can be easily exported to the appropriate tables for TAME so the students' final scores can be calculated with style and late deductions factored in. While course usage of TAME is not a requirement for using this framework with a database, it is something that can be incredibly useful given the amount of data that can be generated.

3.3.3 CSV Output

When the destination for the data parsed from the commentary language ends with a "`.csv`" extension, the TestLoader outputs test names and answers with their matching rules and point ratios to a CSV file for use with JUnitE. No other information is output as part of this mode. CSV mode is designed to facilitate grading in situations where a database installation is not available.

In its current form, the CSV output does not contain all of the same data as the database version. However, CSV files are also easily read by spreadsheet programs and can be easily distributed to individuals who require the answer data (graders, TAs, etc.) who may or may not know SQL.

3.4 Test Support Library

The test support library provides convenient interfaces with the pipeline and the data extracted from the commentary language. The main class responsible for driving this interface is JUnitE, a class modeled after JUnit.

3.4.1 JUnitE

JUnitE is responsible for connecting the information gathered from the commentary language with the student results generated by the execution of the test set. JUnitE also allows the test set to send new information to the pipeline to be stored. For example, it can look to see if a student's answer of 59.75 matches any of the predefined answers for that specific test or record that the student's linked list contains seven nodes.

3.4.2 Evaluating Answers

The real power of JUnitE comes from its `evaluateAnswer()` method that replaces the assertion libraries used by testing frameworks. This method uses the test name and the student answer to determine if the student's answer matches any of the predefined answers. If a match occurs, the score associated with that answer is recorded in the database. If a match is not found, the student result is flagged with a score of -1 for later review by the grader. Metric data is stored in the database regardless of whether or not the student answer is flagged for review. The metric data can be used to help determine what ratio of points the student answer is worth in the event the answer is flagged.

This method is designed to take in the student answer if it is a Java string, integer, or double value. As an alternative, the student answer can be wrapped in a custom object that inherits from a class built as part of JUnitE, FileAnswer. All known answers from the data source are compared to the student answer. Equality is the default comparison for numeric and string values. For Java

strings, the test information can specify a variety of string options including case and space parameters as well as arbitrary regular expressions and their respective matching flags. These rules are governed by each answer and are indicated in the creation of the test file that is sent to the TestLoader program.

Additionally, with the FileAnswer class, it is possible for the instructor to define a custom comparison between the student's answer and the predefined answers by extending the class and overriding the `compareTo` method. By default, the FileAnswer `compareTo` is the Java string implementation, but it is capped so that it can only return a value between zero and one-hundred. This number represents the distance away from the expected result the student answer is. Within JUnitE, this number is subtracted from one-hundred. The difference is then divided by one-hundred to arrive at a decimal ratio which is the amount of credit the student's answer is worth. This comparison can be adjusted to ignore, multi-spaces, all spaces, and/or casing.

3.4.3 Metric Data Capture

A metric is a floating point numerical value with a label that defines what the number represents. Each metric is associated with a specific test and rubric and this information is used by the clustering algorithm to create logical groupings of students whose answers were not predefined. These groups can then be displayed to the grader and could help determine an appropriate and consistent score for wrong answers.

3.4.4 Exception Management

While the testing framework catches any exceptions thrown by student code during execution and prevents them from halting the testing process, it may also prevent the student answer from being recorded. To handle this, JUnitE includes a method called `logExceptionThrow()`, which allows the instructor (should they so choose) to use a try/catch

block to capture the exception thrown by the student's code during a particular test and record this as their result along with a point ratio the instructor would like to award the student.

3.4.5 Record Mode

While the commentary language allows the instructor to describe tests using a set of Javadoc-like tags, it is also possible to generate a set of expected answers by executing a sample solution (or multiple sample solutions) against the test set. This is something that JUnitE supports this through the *Record Mode* feature. The instructor simply calls the static method `JUnitE.setRecordMode()` in JUnit test file, which redirects the methods used in evaluation to instead write to the data source. JUnitE also includes an overloaded `evaluateAnswer()` method that takes in all of the fields necessary to specify an answer record in the data source.

3.5 SimilarityLib

SimilarityLib provides a useful set of methods to simplify the task of comparing the student answer to the correct answer. It provides support for comparing numerical values, strings, data structures, and output files.

3.5.1 Comparing Numeric and String Variables

SimilarityLib supports comparing numerical data types against each other and against an optional epsilon tolerance. Number comparison extends the Java `Number.compareTo()` semantics, by returning values that can be positive, negative, or zero. When comparing numbers with an epsilon tolerance, the result is zero if the difference is less than the epsilon. In principal, the magnitude of a non-zero value represents the magnitude of difference between the student answer and the sample answer.

For strings, SimilarityLib provides three key functions: removing user-specified keywords before comparison, removing spacing (either multiple spaces or all spaces) before comparison, and also comparing strings with case and/or space matching enforced or not enforced.

3.5.2 Comparing Data Structures and Nodes

Comparing data structures is a two step process. First, the data structure(s) must implement specific interfaces to support comparison for both the structure itself and its nodes. Using interface based polymorphism, SimilarityLib communicates with the data structure or data structure nodes to run the desired comparison (such as comparing the contents of two nodes). SimilarityLib is able to compare data structures to determine whether two data structures contain the same nodes or if the structures themselves are the same. Nodes can be tested to see if they are equal and can be compared to one another.

3.6 GradingKit

The GradingKit library provides a Java API for accessing key database information without requiring explicit SQL queries. When developing a database driven application the programmer is required to know SQL and have intimate knowledge of the structure and relationship between entities in the specific database being used. GradingKit provides a far simpler interface that allows the test program writer to access the grading database. Using Model-View-Controller (MVC) to represent table entries and handle database operations means that GradingKit can build and execute the SQL queries for the programmer and maintain database integrity. In doing so, it reduces the amount of overhead necessary for developing new applications for the pipeline.

GradingKit handles all JDBC related objects making them invisible to the programmer. This becomes especially beneficial when dealing with ResultSet objects which can (depending on the RDBMS in use) lock the database or tables even after the ResultSet object has gone out of

scope. By having GradingKit handle the JDBC objects, the programmer does not have to worry about this issue. GradingKit opens the ResultSet, extracts the data into Java objects the developer can work with and closes the ResultSet instance. All the developer ever sees is the objects representing their data.

The sample code segment in figure 7 shows how GradingKit can be used to group students who arrived at the same answer.

```
75    private void createRubricMetricClusters( Test test, TestRubric rubric,
76            String metric )
77    {
78        StudentResultController srController = new StudentResultController();
79
80        Vector<StudentResult> results =
81                srController.getStudentAnswersByTestAndRatio(
82                        rubric.getItemName(), rubric.getVariant(),
83                        rubric.getTestName(), -1 );
84
85        HashMap<String, Vector<StudentResult>> clusters = new HashMap<String,
86                Vector<StudentResult>>();
87
88        for( StudentResult sr : results )
89        {
90            if( !clusters.containsKey( sr.getStudentAnswer() ) )
91            {
92                clusters.put( sr.getStudentAnswer() ,
93                        new Vector<StudentResult>() );
94            }
95
96            clusters.get( sr.getStudentAnswer() ).add( sr );
97        }
98
99        this.saveCluster( test, rubric, metric, clusters);
100   }
```

Figure 7: Sample code using GradingKit

On line 78, a `StudentResultController` object is instantiated and then is used to get a list of students for a particular item (assignment), variant, and test combination that had their answers flagged for review (represented by the -1). Code then iterates over these students to populate a Java `HashMap` then saves it.

Both the `Test` and `TestRubric` objects as well as the `StudentResultController` are all parts of GradingKit. By using these objects, the programmer is able to interact with the data stored in the database in an object oriented manner. The programmer only needs to focus on the Java code and does not have to interact with the database directly through SQL or with the JDBC driving the database connection. Should the RDBMS in use change, the code in the figure above will not be affected since the database interaction has been abstracted away.

CHAPTER 4

IMPLEMENTATION

4.1 Overview

There are two major processing pipelines in the system: assignment specification and assignment testing. The assignment specification step defines the assignment solutions, the tests to be run, the expected answers for those tests, and the credit to be awarded for each known answer. This pipeline is shown on the left side of figure 8 and consists entirely of the TestLoader program that parses the TestFile and stores test and answer data information into the data source. The testing pipeline shown on the right side of figure 8, loads and executes the tests and the student programs using the information in the data source to match expected answers to student answers.

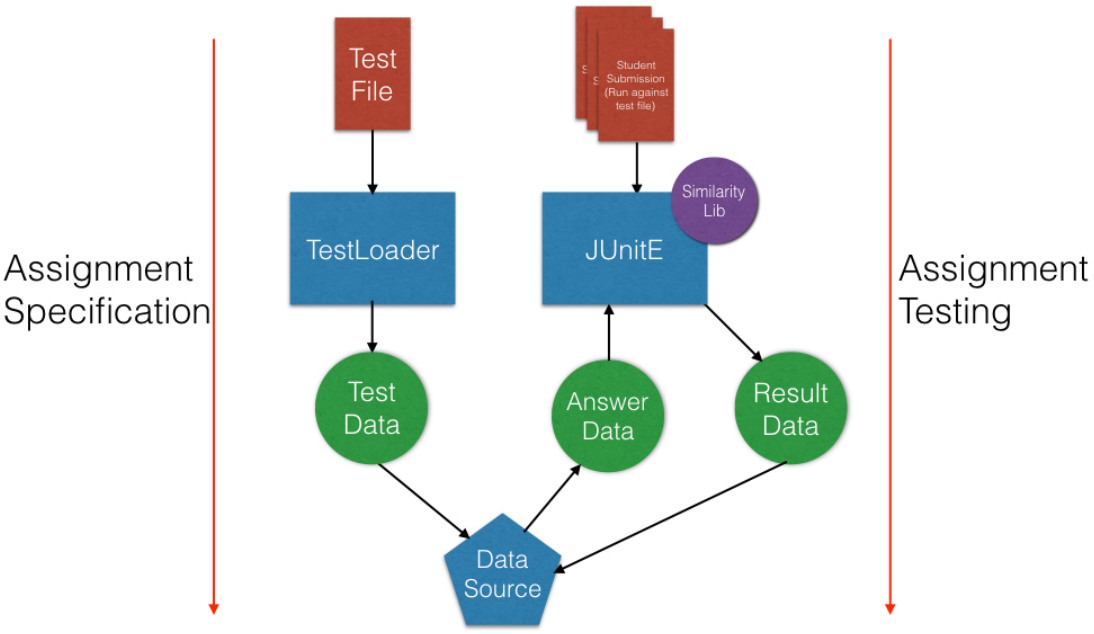

Figure 8: The RPM pipeline

4.2 Data Management

The pipeline utilizes two mechanisms to handle the data it contains. Currently, a SQLite database is used for storage of all information. The commentary language links data from the testing code to the testing environment.

4.2.1 Database

Given the amount of data generated as part of the pipeline, a relational database is the logical choice for storage. In order to facilitate interactions with the TAME grading software, the table structure for the pipeline is stored in the same schema/file as the table structure for TAME. The database has tables for tests, answers, student results, metrics, clusters, and the TAME rubrics. Table 1 (below) lists each database table and the data that it holds.

Table	Description of Data
tbl_assign	Contains all of the assignment names and their description.
tbl_test	Contains all of the tests and subtests for each assignment as well as their min group size and epsilon for clustering.
tbl_test2Rurbic	Links tests to TAME rubrics.
tbl_testAnswer	Contains all of the answers for the tests as well as the matching rules.
tbl_code	Contains the text from the JUnitE test file uploaded from the TestLoader.
tbl_studentResult	Contains the answers generated by student code and the amount of credit those answers are worth (or -1 if they need to be manually evaluated).
tbl_studentMetrics	Contains collected metric data from running student code.
tbl_resultCluster	Contains the clusters of flagged student results.

Table 1: List of tables used in the RPM pipeline

4.3 TestLoader Class Strucutre

The TestLoader program contains models to contain the data found in the commentary language. A single controller class conducts the parsing. Table 2 summarizes of the classes and interfaces used in the TestLoader program.

Package/Class or Interface	Purpose
models/Answer	Holds a specific answer for a test.
models/HeaderData	Holds the assignment id, variant id, and assignment description.
models/Rubric	Specifies rubric's section number and line number.
models/Subtest	Holds information about a particular subtest.
models/Test	Holds all data for a specific test.
models/TestRubric	Gives the rubric a weight against the test.
controllers/TestUploadController	Main class for the test loader, responsible for conducting the parse and sending it to be outputted.
connection/OutputConnection	Generic interface for all output connections.
connection/CSVOutputConnection	Outputs to a CSV file, implements the OutputConnection interface.
connection/DBOutputConnection	Outputs to a SQLite database, implements the OutputConnection interface.

Table 2: TestLoader classes and interfaces

4.4 JUnitE Class Structure

JUnitE's design takes advantage of interface-based polymorphism to allow the instructor to specify how answer data enters the system, graded student results exit the system, and also the mode in which JUnitE is running (record versus test). The polymorphism of the FileAnswer class allows the instructor to implement custom comparisons between the solution and the student code. Table 3 summarizes the classes in JUnitE.

Package/Class or Interface	Purpose
models/AbstractAnswer	Generic answer class, parent to FileAnswer.
models/FileAnswer	Allows the instructor to create a customized comparison function to evaluate a student answer against their expected answer, child of AbstractAnswer.
inputConnections/InputConnection	Interface any class looking to serve as a connection to input answer data must implement.
inputConnections/DBConnection	Class to load answer data from SQLite, implements InputConnection.
inputConnections/CSVConnection	Class to load answer data from a CSV file, implements InputConnection.
main/JUnitE	Main class for the JUnitE package.
main/RunMode	Interface that any class looking to serve as a mode in which JUnitE can run in must implement.
main/RecordMode	Implementation of the RunMode interface allowing for answers to be recorded.
main/TestMode	Implementation of the RunMode interface allowing for student results to be compared against predefined answers.
outputConnections/Writer	Interface that any class looking to serve as an output stream must implement.
outputConnections/DBOutputWriter	Outputs data to an SQLite database implements Writer interface.
outputConnections/CSVOutputWriter	Outputs data to a CSV file, implements Writer interface.
utils/ArrayUtils	Utility for converting a string array to a string.
utils/IDRetrievalTool	Utility for getting a student's ID from the TAME file system.
utils/PrintRedirect	Utility for capturing information being sent to standard output and standard error streams.

Table 3: JUnitE classes and interfaces

4.5 SimilarityLib Class Structure

A key motivation behind the design of SimilarityLib is to make it easy for users to implement customized data comparison functionality appropriate for a specific type of

programming assignment. Table 4 details the classes and interfaces found in the SimilarityLib package.

Package/Class or Interface	Purpose
interfaces/NodeCompare	Interface that any data structure node class looking to be used with SimilarityLib must implement.
interfaces/StructureCompare	Interface that any data structure looking to be used with SimilarityLib must implement.
similarity/VariableTests	Similarity tests for numeric and String values.
similarity/StructureTests	Similarity tests for data structures.
similarity/MotionTest	Similarity tests for motion / animation.

Table 4: SimilarityLib classes and interfaces

CHAPTER 5

RESULTS

5.1 Batch Program Support

Although many of the programs that are used as part of the curriculum at UNH are interactive and/or graphical in nature, there are still components and data data structures driving them that can be tested in a batch manner.

5.1.1 CS 416 Spring 2015 Laboratory 11

This laboratory assignment is the students' first exposure to linked list data structures. They must use recursion and their knowledge of the structure of linked lists to perform the following tasks:

- Iteratively compute the average of the numbers stored in the list
- Recursively compute the average of the numbers stored in the list
- Use iteration to reverse the list
- Use recursion to reverse the list.
- Compress the list based on the key values of the nodes.

Students do not need to implement the linked list themselves nor do they need to generate test data.

5.1.2 Test Set Design and Sample Data

The test set is modeled on a program written by the instructor to exercise the student code. The test set consists of one test for each task (except for the recursion reversal of the list, which has been broken into two tests) and tests the student's solution against three different linked lists.

In total each student's code is subjected to eighteen tests. The sample data for the run is the student code base from the Spring 2015 version of CS 416.

5.1.3 Test Result (Batch Phase)

Group	Number of Tests (Total Tests: 1,656)	Percentage (Out of 1,656)
Full Credit Awarded	1,360	82.13%
Less than Full Credit Awarded	201	12.14%
Unpredicted Answers	89	5.37%
Answer Missing	6	0.36%

Table 5: Results of running 11L

As table 5 shows, over eighty percent of tests successfully matched the correct answer and twelve percent matched predefined answers meriting partial credit. In total, this means that over ninety-four percent of the tests for this assignment were automatically graded.

The six tests for which results are missing occurred because the student code entered an infinite cycle (infinite loop, recursion, etc.) and JUnitE terminates a test if its execution exceeds an instructor specified limit.

5.1.4 Test Result (Clustering)

To help the grader we want to cluster unpredicted answers that are similar. In the test run, there were eighty-nine unpredicted answers spread over fourteen tests. We have chosen one of those tests with ten unpredicted answers to demonstrate how clustering can be used to facilitate grading. The input to the student code is a linked list of nodes that contain a String id field and an integer value field; the id fields need not be unique. The student code is supposed to "compress" the list so that there is just one node for each unique id and that node's value field must be the sum of the value fields in the original list for that id. The list class provided to the student has a `toString` method that returns the list as the node `toString` results ("id:value") in order, each followed by ",". The expected result is "ie:100, bi:283, ik:123,". It is possible to cluster the

students who did not arrive at this answer using their answers as Strings or using a predefined metric, such as the number of nodes in the linked list.

	Student Answer	Number of Students
Cluster 0	ie:106, bi:184, ik:123,	4
Cluster 1	ik:123, bi:283, ie:100,	2
Cluster 2	ie:6,	1
Cluster 3	ie:6, ie:6, bi:48, bi:73, bi:63, ik:34, ie:62, ik:89, ie:32,	1
Cluster 4	ie:32, bi:99, ik:123,	2

Table 6: Clustering results by exact equality

As shown in table 6, by clustering the unexpected answers using the answer itself, five clusters are generated. This requires the grader to make only five grade determinations instead of ten and guarantees that the same answers get the same grade which is not easily accomplished with the amount of grading necessary for each assignment in CS 416.

Although useful, exact match clusters have limits: each unique answer is another cluster. Alternative clustering options often can produce more effective clustering. In this case, for example, we could try clustering by a simple list metric, the node count. The algorithm retrieves a list of the students whose answers are flagged for review based on a specific test ordered by the metric value. All students who arrived at the same metric value are placed in the same group. For this test, there are exactly three unique keys, so the resulting list should have three nodes. A three node list with the three unique keys identifies programs that compressed correctly, but computed the wrong values. This leads to a large cluster that is likely to warrant similar (if not identical) credit. Table 7 shows the clusters for this metric for this test.

	Student Answer(s)	Students in Cluster
Cluster 0	ie:106, bi:184, ik:123,	8
	ie:32, bi:99, ik:89,	
	ik:123, bi:283, ie:100,	

Cluster 1	ie:6,	1
Cluster 2	ie:6, ie:6, bi:48, bi:73, bi:63, ik:34, ie:62, ik:89, ie:32,	1

Table 7: Clustering results by distance clustering

5.2 Interactive/Graphical Program Support

Because of the importance of interactive and graphical programs to the curriculum at UNH, we evaluated the effectiveness of the Abbot [6] tool for testing AWT and Swing applications and provided appropriate interface that it to be used as part of the automated grading with JUnit.

5.2.1 Assignment 4L

We used assignment 4L from the CS 416 curriculum to test Abbot in this context. In the assignment, students need to make a snowman drawn in SWING bounce around inside a frame. The student's must create a set of labeled buttons at the top which control the speed of the snowman as well as the timer that triggers his animation.

5.2.2 Test Design

Each test in the test set uses Abbot functionality to test one of the buttons in the UI and validate that it has the correct effect on the snowman or the animation timer. A separate test is needed to verify program correctness that depends on a specific sequence of button presses to occur.

5.2.3 Execution Outcome

Executing the test shows that it is in fact possible to have Abbot test the graphical components of a CS 416 program. Abbot is able to interact with the various graphical components on the screen and simulate events typically issued by the mouse and keyboard by a human grader.

CHAPTER 6

FUTURE RESEARCH

6.1 Additional Testing

Although we have validated that the current software does work and is capable of successfully grading student work, this testing is by no means exhaustive. More testing needs to be conducted to examine the full capabilities of the software across a wider range of assignments and labs. This testing needs to involve both instructors and graders in order to assess how well the clustering algorithms assist them in determining partial credit for unpredicted results.

6.2 Additional Clustering Algorithms

The simple clustering algorithms implemented in the prototype have limited power. The software architecture supports inclusion of alternative clustering algorithms. We need to implement these and evaluate how effective they are at simplifying partial credit grading.

6.3 Testing Data Structure Capabilities

Although we have shown that JUnit can provide effective support for linked list data structure assignments, there are more complex data structures the software needs to be able to handle if it is going to be useful in a course such as CS 416. We need to implement and evaluate tools for supporting non-linear data structures such as binary search trees, state/game trees, quadtrees, and simple graphs.

6.4 Graphical Support for Commentary Language and Cluster Reorganization

Although the commentary language provides the ability to define all of the necessary information about a test, it is not very user friendly. Although the record mode in JUnitE brings a somewhat more user friendly experience, it requires that the entire test file be completed before

the test data can be recorded. It would be most convenient for the instructor to have a web-based GUI to define the test data.

Additionally, the clustering can benefit greatly from a graphical tool to assist the grader. While additional clustering algorithms may prove to be more adept at creating usable clusters, it may be useful to reorganize those clusters and merge/split clusters.

APPENDIX A

This appendix serves to demonstrate the proposed workflow for the system in two scenarios. The first is a situation in which the program being tested has a graphical interface component that the students were responsible for implementing given a predetermined list of labels and also connect to various methods within their code. The second, details a program that has no graphical component and just simply prints output to the terminal or a file.

7.1 Part 1: Sample Graphical Program: 4L - Button Events

For this lab, students are asked to create a set of buttons with labels that have been predefined for them. These buttons cause a snowman bouncing around the screen to change in various ways: the "stop" button causes the snowman to stop moving; "faster" causes it to move faster, etc. Students are required to build a border layout and place the buttons with specified labels in the "south" part of the layout and snowman in the "center". They must then connect the buttons to button listeners and implement the necessary code within the listener to make the desired action happen. In some cases, (e.g., the "faster" and "slower" buttons) this also requires the modification of instance variables in the snowman object.

Grading this program can be difficult for the grader. For each student the grader must run the student submission which can be especially tedious if they have a slow connection to the server. Furthermore, once the program is running, the grader has to click on each button to confirm the functionality. While this is not overly difficult to do for a few student programs, as the number of programs grows, the time increases at a rapid rate. Automating even part of the testing would reduce the grading time considerably.

7.1.1 Test Preparation

With the proposed system, the Abbot library can be used to automate the testing of the graphical components. For example, appropriate JUnitE documentation comments can specify the name, max point value, and any desired full / partial / no credit answers for a test method created for this lab. The test method instantiates an instance of the application and uses the Abbot library (included with JUnitE) to build an instance of the Matcher class to find the button labeled "stop". Abbot can then simulate a click event on that button. Once this is done, the JUnitE test can invoke an application method to see if the timer that controls the snowman's animation has been stopped. The result of this method call is sent to JUnitE for evaluation and for points to be assigned to the student for that particular task.

A similar specification is required for each test desired, using Abbot to find any graphical items on the screen needed for the test and using a combination of Abbot functionality and methods to assess the student code. JUnitE handles the assignment of points based on how the student result compares with the predefined answers.

7.1.2 Batch Run

When the instructor runs the TAME pregrade script, each predefined test case is run and creates a new instance of the student program, finds the items on the interface, and interacts with them as specified in the test. Once the test code has evaluated the student program, the results sent to JUnitE to be compared to the pre-defined answers. Any answers JUnitE cannot match are flagged for review by the grader.

7.1.3 Benefits of Automated Grading in the Given Scenario

Abbot is able to assess a student's submission for this assignment in less than twenty seconds whereas it takes a human grader upwards of five minutes per student. While it may still

be necessary for the human grader to do some amount of evaluation in the Post-Batch phase, the amount of work required by them here is much less than what is be required if they needed to grade each student manually.

7.2 Sample Command Line Program: 6L - Practice with Recursion

For this lab, students fill in a set of four small recursive methods each looking to accomplish a different task. For example, they must write a method to calculate a factorial and a method to determine if two strings are palindromes. Unlike 4L, this lab is all text based; there are no graphical components to it. The grader compares each student's output with the output from a sample solution and determines out whether and how many points to deduct. It is important that the same or similar errors receive the same or comparable penalties.

7.2.1 Test Preparation

Although the Abbot library is not needed for this assignment, the instructor must still define the documentation comment tags and the test specifications for each test, so JUnitE has the data it needs to make the comparisons when called. In each test, the JUnitE methods evaluate success or failure or partial failure.

7.2.2 Batch Run and Post-Batch Run Analysis

The biggest problems in this lab are those that arise from infinite recursion. Because JUnit has the ability to have a specified time limit for a test, it is easy to prevent a test from running infinitely. Additionally, should an infinite recursion results in a Stack Overflow Exception, JUnit can catch and handle this issue. In these cases, while nothing shows up in the block view when the grader arrives at that student in TAME, no points are assessed to that specific rubric and the grader can investigate the problem further.

7.2.3 Benefits of Automated Grading in the Given Scenario

The problem with this lab is that all of the text output must be compared to the sample output for each student. Having all of this work automated reduces the chance for errors in assessing student submissions against the expected answers. Additionally, since non-correlated answers are put into groups, when the grader assesses these groups and assigns a point value to their answer, the same point value gets assigned for that problem to all students in the group which reduces the chances of inconsistent grading.

BIBLIOGRAPHY

[1] X. Sun, "Grading System Manual (For Course Administrators)," University of New Hampshire Department of Computer Science, Durham, 2013.

[2] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT," *SIGCSE Bulletin,* vol. 40, no. 3, pp. 328-328, 2008.

[3] J. Stegeman, "Unit Testing Your Application with JUnit. Oracle ADF Development Essentials," Oracle. [Online]. [Accessed 6 April 2015].

[4] V. K. Proulx, "Test-driven design for Introductory OO programming," in *ACM SIGCSE Bulletin*, Chattanooga, 2009.

[5] C. Beust and H. Suleiman, Next Generation Java Testing: TestNG and Advanced Concepts, Boston, MA: Pearson Education, 2007.

[6] T. Wall, "Abbot Java GUI Testing Framework," Timothy Wall, 2011. [Online]. Available: http://abbot.sourceforge.net/doc/overview.shtml. [Accessed 1 December 2014].

[7] V. K. Proulx and W. Jossey, "Unit test support for Java via reflection an annotations," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, Alberto, 2009.

[8] A. Ruiz and Y. Wang Prince, "Test-Driven Development with TestNG and Abbot," *Software, IEEE,* vol. 24, no. 3, pp. 51-57, 2007.

[9] T. Allevato and S. H. Edwards, "The Web-CAT Community," The Web-CAT Community, 24 November 2014. [Online]. Available: http://web-cat.org/group/web-cat. [Accessed 1 December 2014].