A GENERIC AND EXTENSIBLE TOOL FOR MARKING PROGRAMMING ASSIGNMENTS

G. Tremblay, F. Guérin and A. Pons
Dépt. d'informatique, Université du Québec à Montréal
C.P. 8888, Succ. Centre-ville
Montréal, Qué. Canada
email: tremblay.guy@uqam.ca

ABSTRACT
Marking programming assignments in introductory programming courses involves a lot of work: each program must be tested, the source code must be read and evaluated, etc. With the large classes encountered nowadays, the feedback provided to students through marking is thus rather limited, and often late.

Tools providing support for marking programming assignments do exist, ranging from support for administrative aspects through automation of program testing or support for source code evaluation based on metrics.

In this paper, we introduce a tool that provides support for submission and marking of assignments. It aims at reducing the workload associated with the marking task and, more importantly, at providing *timely* feedback to the students, including feedback *before* the final submission. Furthermore, the tool has been designed to be *generic* and *extensible*, so that it can deal with programs in various languages and it can be extended with various marking components (modules).

KEY WORDS
Educational Software and Hardware, Automated Marking, Introductory Programming, Unit Testing

1 Introduction

Marking computer programs in introductory programming courses is, and always has been, a lot of work, as it involves dealing with many aspects. First and foremost, the program must be tested to ensure that it exhibits the correct behavior. In addition, the text of the program (source code) and its accompanying documentation must be read in order to evaluate the program structure and style and to ensure that the appropriate standards have been adhered to.

With the large classes encountered nowadays, marking is generally done with the help of teaching assistants (TAs). These assistants may be graduate students or even advanced undergraduate students. The feedback they provide to students may thus be somewhat limited and, more importantly, may come late in the students' learning process, as the marking process for large classes may be rather lengthy. For instance, when a student finally receives her graded assignment, the topic dealt by this assignment may already be "outdated", thus less significant. Furthermore, if the student made a significant error, it is too late to correct it. In other words, the typical approach to marking programming assignments requires a lot of effort from the instructors and/or TAs, yet provide little *timely* feedback to the students.

Tools that provide various forms of support for marking programming assignments do exist, some of which will be presented in more detail in Section 2. Such tools may provide support for dealing with the administrative aspects (submission and management of assignments) as well as support for other tasks associated with marking programming assignments, ranging from script-based execution of test cases [14, 13] to metrics-based evaluation [10, 16].

In the present paper, we introduce a tool, called Oto, that provides support for the submission of programming assignments as well as for their marking. One of its key goals is to reduce the workload associated with the marking task. In addition, it aims at providing timely feedback to the students, including feedback *before* the final submission deadline. Finally, it has been designed to be *generic*—independent of any specific programming language—and *extensible*—designed so that various forms of "marking" can easily be incorporated.

Underlying pedagogical approach

The design of Oto has been influenced by a specific educational context, namely, the teaching of *introductory* courses in object-based programming. [1] Some of the pedagogical principles underlying such courses are the following:

- In a first programming course, the emphasis should be on *programming-in-the-small*. This means students should mostly be required to implement specific modules (viz., classes), for which the instructor provides the appropriate specification (viz., interfaces), not large-scale programs.

- The students should be introduced early to the key practice of separating the presentation code from the application logic. In other words, the emphasis should

[1] We say "object-based", as opposed to "object-oriented", because the notions of object and class are introduced early, but inheritance *is not*.

be on developing classes implementing some appropriate domain model, not on writing detailed I/O routines or developing GUIs.

- The important role of unit testing in developing quality software should be stressed.

Outline of paper

The paper is organized as follows. First, in Section 2, a number of existing tools for dealing with programming assignments are presented. In Section 3, the key features of JUnit, a unit test framework for Java, are presented. Section 4 then presents the key functionalities of the Oto marking tool. How genericity and extensibility is achieved is then described in Section 5. Finally, in Section 6, some implementation aspects of Oto are briefly described, followed by a conclusion together with future work.

2 Existing Tools for Marking Programming Assignments

A wide variety of tools that provide support for dealing with programming assignments have been developed over the years. The key features of these tools can be classified into three major categories:

1. Management of assignments: These tools support the various administrative tasks that must be handled by the instructors and TAs: receiving the students' submissions, keeping track of marks, sending feedback (grade, marking report) to the students when the assignments have been marked, etc. [7, 14].

2. Evaluation of correctness: The usual way to assess the correctness of programs is through testing. A key goal is to automate as much as possible the testing process, as it can be quite tedious given a large number of submissions.

   Different tools support various forms of testing. For instance, assuming the programs use textual I/O, testing can be as direct as using a strict textual comparison (for example, using "`diff -bB`" in Unix). Assessing the correctness of a program output can also be more subtle. For example, the *expected* output can be described using a context-free grammar; the output produced by a program can then be parsed to ensure it complies with the grammar specification [14].

3. Evaluation of quality: The "quality" of a program is definitely an elusive notion. Program quality can be evaluated, among other things, by examining the program structure (e.g., using appropriate source code complexity measures, including coupling and cohesion) or the programming style (e.g., proper indentation, use of symbolic constants, choice of identifiers, presence of internal documentation). Some of these properties can be evaluated from the source code with the help of static program analysis, based upon appropriate design metrics [10, 16].

These three categories, of course, are neither all encompassing, nor mutually exclusive, so a given tool can exhibit features from many categories. For example, in the late 80's, the TRY system [18] allowed students to submit their programs, and then allowed instructors to test the submitted programs, tests which evaluated the program outputs on a purely textual basis (modulo blank spaces and lines).

The ASSYST [14, 13] system allows students to submit their assignments by email. The instructor later tests and marks the submitted programs, and then sends back an evaluation report to the students. Marking is done through partially automated testing (based on a context-free grammar specification of the expected output) as well as by using a number of metrics to evaluate the quality of the source code, the efficiency of the resulting program, as well as the effectiveness of the tests developed by the students to test their own programs.

The BOSS system [17, 15] supports both submission and testing of programs. Testing is based, again, essentially on comparing textual output.

Curator [12] is a more recent offering that relies on modern web technology. It can be used for various kinds of assignments, not only for programs. Automatic testing of programs is supported, though again it is based on a strict textual comparison.

There are two major disadvantages with testing based on textual comparison of program output. First of all, this generally makes the testing process quite *strict*. For instance, the student documentation for Curator indicates that "It is important that your output file not contains any extra lines or omit any line" [12].

More importantly, such an approach requires putting a lot of emphasis on producing program output through console I/O, clearly a secondary aspect for an object-based approach that attempts to separate presentation from application logic.

The OCETJ tool [20] was developed with the goal of avoiding reliance on text-based (console I/O) testing. OCETJ supports both the submission of programming assignments (in Java) by students and the automatic testing of the submitted programs. Testing is done using the JUnit framework [5], a tool that supports the automatic execution of test suites and test cases, a key practice of professional software development [11] which is also becoming prevalent in educational context [2]. OCETJ was also designed with the goal of providing quick feedback to the students, which is done through the use of public vs. private test suites—this is explained in more detail in Section 4. OCTEJ's implementation, however, was far from generic, as it provided support only for handling Java programs and for their testing with JUnit. Furthermore, its implementation was intimately tied to the platform on which it was developed (intra-net from Cégep du Vieux-Montréal).

In the following sections, we describe the Oto tool, which was designed with the goal of supporting all aspects of programming assignments marking and the goal of providing early feedback to students. Furthermore, since it does not rely on testing through textual output but instead relies on test unit technology, we briefly introduce the JUnit framework in the next section.

3 The JUnit Testing Framework

Although the importance of testing has long been recognized, proponents of agile methods have recently emphasized the beneficial role of unit testing and test automation [3]. Test automation, which allows for the automatic execution and verification of large number of test cases, is neither new nor specific to agile methods [8]. What is new to agile methods is the tight integration of test automation with a test-first approach to code development, allowing testing to be done both early and often [4].

For such an approach to software construction to be feasible, appropriate tools for automating the testing process must be available. One well-known such tool is JUnit [5], a unit testing framework for Java.

```
class Account {
  private Customer cstm;
  private int bal;

  public Account( Customer c, int initBal )
  { cstm = c; bal = initBal; }

  public int balance()
  { return( bal ); }

  public Customer customer()
  { return( cstm ); }

  public void deposit( int amount )
  { bal += amount; }

  public void withdraw( int amount )
  { bal += amount; }
}
```

Figure 1. `Account` class to be tested (with an error in method `withdraw`)

Suppose the class presented in Figure 1, a simple class defining bank account objects, is to be tested. Note that the `withdraw` method contains an error (a "`+`" has been used instead of a "`-`", a typical copy-paste error). A JUnit class for testing the `Account` class is presented in Figure 2. The methods whose names start with "`test`" represent specific *test cases*. These test cases use `assertEquals` to check whether the result returned by the method under test (second argument of `assertEquals`) is the expected one (first argument). Other variants of `assert` methods do exist, for example, `assertTrue`, `assertNotNull`.

A key feature of `assert` methods is that they generate no output *unless* the expected condition is *not* satisfied. Whenever this occurs, an `AssertionFailedError` is thrown. Within the test framework's context, this exception is then caught and an appropriate message is written to the test log.

```
public class AccountTest extends TestCase {
  public AccountTest( String name ) {
    super(name);
  }

  public void testBalance() {
    Account acc = new Account( new Customer("Joe"), 100 );
    assertTrue( acc.balance()  == 100 );
  }

  public void testTransfer() {
    Account acc = new Account( new Customer("Joe"), 100 );
    int initBal = acc.balance();
    acc.deposit ( 50 );
    acc.withdraw( 50 );
    assertEquals( initBal, acc.balance() );
  }

  public static Test suite() {
    return new TestSuite(AccountTest.class);
  }

  public static void main( String[] args ) {
    junit.textui.TestRunner.run( suite() );
  }
}
```

Figure 2. `AccountTest` class for testing `Account`

A collection of test cases is called a *test suite*. Each test class must define its associated test suite, as done by method `suite()` in Figure 2. In this case, all methods whose name start with "`test`" get implicitly included in the test suite. The resulting test suite can then be executed as shown in method `main`, thus generating the following output:

```
There was 1 failure:
1) testTransfer(AccountTest)
    junit.framework.AssertionFailedError:
    expected:<100> but was:<200>
at AccountTest.testTransfer(AccountTest.java:19)
at AccountTest.main(AccountTest.java:27)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

4 Key Functionalities of Oto

Use case 1 presents Oto's high-level use case—what Cockburn would call the *summary business use case* [6]. In order to use Oto for a specific assignment, the instructor must first define an appropriate *evaluation*. Such an evaluation generally includes two components (any of them is optional):

- A *preliminary verification* that acts as a *filter* to ensure that the students' submissions are minimally correct with respect to the requirements. It is this preliminary

Use case # 1 Define and mark an assignment
Scope: University.
Level: Summary.
Actors: Instructor, Students, Teaching assistant(s).
Preconditions: The instructor teaches a course and wants to have an assignment marked by Oto.
Main Success Scenario :

1. The instructor writes up the assignment and designs the evaluation scripts that will be used to verify and mark the students' assignments.
2. The instructor defines an evaluation.
3. Oto confirms the creation of the evaluation and makes the evaluation scripts thus defined available to the students.
4. The students verify their solution (to obtain preliminary feedback).
5. Before the final due date, the students submit their assignments and Oto saves the submitted assignments.
6. The instructor (or teaching assistant) marks the assignments that were submitted by the students, using the appropriate evaluation script.
7. The instructor uses the report produced by Oto as one input (along with the hard copy program listing and documentation) to determine the final marks.

Use Case 1: Summary use case for Oto

verification that provides students with early feedback on their solution, thus providing a *public* suite of test cases.

- A *final verification* that is used to evaluate and mark the submitted assignments, providing a *private* suite of tests.

Before the final submission date, students can perform preliminary verifications of their assignments, in order to receive feedback and ensure they are "*on the right track*". Such submissions are not saved by Oto. On the other hand, students can also submit their final solution, in which case the submitted assignments are saved for later marking.

Finally, once the final submission date is reached, the instructor (or its surrogate) can perform the final marking of the submitted assignments, as specified by the final verification component of the evaluation.

Each (component of an) evaluation is described by various attributes, e.g., which course and assignment it pertains to, its starting and ending dates. More importantly, each (component of an) evaluation is associated with an *evaluation script*, as described in the following section.

5 Marking Scripts and Modules: Genericity and Evolution

Two key design goals of Oto are to make the tool both *generic* and *extensible*. Generic means, first and foremost, not tying it to a specific programming language, that is, making it possible to use Oto for marking programming assignments written in various languages. Extensible means not tying it to a specific form of evaluation. Thus, although any evaluation will most certainly include a testing component, how this testing will be done may vary depending on the programming tools and environment.

More importantly, it should be possible to extend the marking tool to include additional types of evaluation. For example, we have identified a number of aspects which could be covered by marking tools, any of which could be evaluated through an appropriate marking component [9]:

- Quality of the code, as determined by static measures.
- Efficiency of the program, as evaluated through dynamic measures.
- Correctness of the program, as evaluated through test suites and cases.
- Quality of the tests developed by the students to test their own program.
- Originality of the code, i.e., plagiarism detection.

The goals of genericity and extensibility are attained using *evaluation scripts* and *evaluation modules*.

5.1 Evaluation scripts

An evaluation script—also called an OtoScript—is similar (in spirit) to a `makefile` in that it describes the various tasks that need to be accomplished in order to evaluate a student's assignment. An OtoScript contains a sequence of declarations defining constants, file names, intermediate results, or tasks to be performed. An example script is presented in Figure 3.

This script first introduces a symbolic constant (`maxMark`) together with various symbolic names (`assignmentName`, `studentPgm` and `testClass`)—the "`=?`" binding operator ensures that a file with the specified name (right hand side) does exist, otherwise an appropriate error message is generated and the processing of the current assignment is aborted. Two tasks are then specified: one to compile the submitted program, the other to test the resulting compiled program. Note that the latter task will only be executed if compilation succeeded without any errors—more complex preconditions can also be specified using `ensure` clauses. Input arguments to a task are specified using keyword parameters, as are the outputs produced by a task, for example, `test.nbFailures`.

When a student assignment is evaluated using the above script, an *evaluation report* is produced and contains two different types of elements:

```
maxMark = 100

# Assignment submitted by the student.
assignmentName = Assignment1
studentPgm     =? ${assignmentName}.java

# Test class provided by the instructor.
testClass = Assignment1Test

compile :: javac {
  sourceFile = $studentPgm
}

<<Testing of program>>
+test :: junit {
  targetClass = $assignmentName
  testClass   = $testClass
}

<<Final mark (over $maxMark)>>
finalMark = $( $maxMark - 5 * $test.nbFailures )

output { finalMark }
```

Figure 3. An example evaluation script

```
OUTPUT
  Final mark (over 100) = 80

EXECUTION REPORT
  Testing of program:
    Number of tests run: 20
    Number of failures: 4
    Number of errors: 0
```

Figure 4. An example evaluation report for the script of Figure 3

- The explicit output to be produced by the script (`output` clause at the bottom of the script).
- A trace of the script's execution, more precisely, of the variables and tasks which have been annotated as *public* (i.e., explicitly annotated with a "+" prefix). For such items, the immediately preceding text, delimited between "«" and "»", is also included in the output.

An identifier can also be associated with an expression, indicated by "`$(...)`", to be dynamically evaluated. Because OtoScripts are evaluated using an underlying Ruby interpreter [19], such an expression can contain any valid Ruby expression. An example report for the script of Figure 3 is presented in Figure 4.

5.2 Evaluation modules

Scripts can be used to create evaluations for specific assignments. However, scripts cannot be used to introduce new mechanisms by which submitted assignments can be evaluated. Such mechanisms are introduced by defining new evaluation modules.

Contrary to evaluation scripts, which can readily be developed by instructors using the OtoScript notation, evaluation modules will be developed by the Oto tool designers and developers. An evaluation module is a self-contained component that provides a well defined interface, with a number of input arguments and a number of output results.

Currently, all evaluation modules are developed using the Ruby scripting language [19] and must satisfy three conditions. First, the class name must obey an appropriate naming convention. Second, it must implement a `run` method which takes three parameters: a list of arguments generated by the script, a list of environment variables describing where to locate the student's and instructor's files, and finally a list of the intermediate results computed by the script before calling the component. Third, on exit, the `run` method must return a list of named results, to be used by the subsequent part of the script.

5.3 Predefined scripts and modules

In order to use Oto and define their own evaluations, instructors will have to learn to write OtoScripts. To alleviate this task, a number of basic skeleton scripts will be provided, which should make it relatively straightforward for instructors to develop their own scripts. Of course, a number of evaluation modules will also be provided, for instance:

- A module that, using `javac`, compiles Java programs and whose interface consists of the following items:
  - Inputs: `sourceFile`, together with optional inputs (e.g., `classpath`, `targetDir`, `compilingOptions`).
  - Outputs: `targetClass`.
- A module that, using the JUnit framework, tests the resulting compiled Java programs, and whose interface consists of the following items:
  - Inputs: `targetClass`, `testClass`.
  - Outputs: `nbTestCases`, `nbFailures`, `nbErrors`.

6 Implementation of Oto

The Oto system is being implemented as a `Unix` command line utility with a single entry point but many subcommands. However, appropriate *façades* have been defined for each category of users, e.g., a plug-in allows students to submit their assignments directly from the BlueJ environment [1]. Every instructor has a dedicated directory managed by Oto and located in his home directory on the Oto server. This space is used to store the final assignments submitted by the students as well as the scripts created by

the instructor. To ensure security and privacy, the utility is run as a `SGID` (Set Group Id) Oto program. Therefore only Oto and the instructor itself may access the files.

Execution of the scripts occurs locally in the student's or instructor's directory. For instance, when a student uses an instructor's script to evaluate his own assignment, the script is run using the student's identity and rights in the directory where the command was launched, so that the possible side effect remains under his responsibility. Similarly, when the instructor marks the collection of submitted assignments, execution is performed within his own account.

When the instructor publishes an evaluation script, the script is first checked for syntax errors. Then, an intermediate code representation is created and stored for subsequent use. Whenever a student or an instructor invokes an evaluation script, the intermediate code is loaded and interpretation starts. During this process, whenever the script calls for a specific task to be performed, the appropriate evaluation module is located. This module can be one of the predefined modules, or a local one, so the instructor can provide his own modules to be used with his scripts. Whenever an error occurs during script interpretation, the error is caught and reported. Of course, the goal is to be able to verify a group of assignments without aborting the whole job because of a bad assignment. Note that the handling of such group of assignments, for the final marking, is implicit, thus not part of the script *per se*. As mentioned earlier, Oto is implemented in the Ruby script language [19], which makes it simple to locate and load code dynamically, especially components.

7 Conclusion and Future Work

Providing timely feedback to students and reducing the workload associated with testing and marking large number of programming assignments have been the initial motivations behind the development of the Oto tool. Two key pedagogical concerns are also addressed by the use of an appropriate testing unit framework, namely, the need to emphasize, early in the curriculum, the importance of separating the application logic from the details of the presentation (avoiding the typical emphasis on textual I/O) and the key role of unit tests for correct program development.

We are currently preparing to start using Oto in a first year Java course (fall term 2005). More precisely, Oto will first be used for marking (mandatory) laboratory assignments, done in class.

For future work, we would like to implement Oto as a web service, allowing it to be used in a more general context than its current one. We also plan to extend it by defining additional evaluation modules. For instance, we plan to develop a module that will perform and compute various *quality* analysis and metrics, with the aim of providing further input to help the instructor (or teaching assistant) determine the final students' grade. Again, the goal will not be to fully automate the evaluation and marking process, but instead to provide support that will help reduce the marking workload. Finally, we also have plan for a module that would detect potential plagiarism, a plague with which, sadly, most universities are now confronted.

References

[1] D.J. Barnes and M. Kolling. *Objects First With Java : A Practical Introduction Using Bluej*. Prentice Hall, 2003.

[2] E.G. Barricanal, M.-A.S. Urbán, I.A. Cuevas, and P.D. Pérez. An experience in integrating automated unit testing practice in an introductory programming course. *SIGCSE Bulletin*, 34(4):125–128, 2002.

[3] K. Beck. *Extreme Programming Explained — Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[4] K. Beck. *Test-Driven Development — By Example*. Addison-Wesley, 2003.

[5] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[7] K.M. Dawson-Howe. Automatic submission and administration of programming assignments. *SIGCSE Bulletin*, 28(2):40–42, 1996.

[8] M. Fewster. *Software Test Automation*. Addison-Wesley, 1999.

[9] F. Guérin. Correction automatique de programmes. Note interne de recherche, Mai 2003.

[10] S.-L. Hung, L.-F. Kwok, and R. Chan. Automatic programming assessment. *Computers & Education*, 20(2):183–190, 1993.

[11] A. Hunt and D. Thomas. *The Pragmatic Programmer— From journeyman to master*. Addison-Wesley, 2000.

[12] Virginia Polytechnic Institute and State University. Curator: an electronic submission management environment. `http://ei.cs.vt.edu/~eags/Curator.html`.

[13] D. Jackson. A semi-automated approach to online assessment. *SIGCSE Bulletin*, 32(3):164–167, 2000.

[14] D. Jackson and M. Usher. Grading student programs using ASSYST. *SIGCSE Bulletin*, 29(1):335–339, 1997.

[15] M. Joy, P.-S. Chan, and M. Luck. Networked submission and assessment. In *1st Annual Conference of the LTSN Centre for Information and Computer Science*, pages 335–339, Newtonwabbey, OK, August 2000.

[16] R.J. Leach. Using metrics to evaluate student programs. *SIGCSE Bulletin*, 27(2):41–48, 1995.

[17] M. Luck and M. Joy. A secure on-line submission system. *Software — Practice and Experience*, 29(8):721–740, 1999.

[18] K.A. Reek. The TRY system — or — how to avoid testing student programs. *SIGCSE Bulletin*, 21(1):112–116, 1989.

[19] D. Thomas and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, 2001.

[20] G. Tremblay and É. Labonté. Semi-automatic marking of Java programs using JUnit. In *International Conference on Education and Information Systems: Technologies and Applications (EISTA '03)*, pages 42–47, Orlando, FL, July 2003. International Institute of Informatics and Systemics.