Oto, A Generic and Extensible Tool for Marking Programming Assignments

G. Tremblay†*, F. Guérin, A. Pons†, A. Salah†

†Département d’informatique, Université du Québec à Montréal, Montréal, QC, Canada

SUMMARY

Marking programming assignments in 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, called Oto, that provides support for submission and marking of assignments. Oto aims at reducing the workload associated with the marking task. Oto also aims 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 the marking process for a specific assignment can easily be customized and the tool can be extended with various marking components (modules) that allows it to deal with various aspects of marking (testing, style, structure, etc.) and with programs written in various programming languages.

KEY WORDS: Educational Software, Automated Marking, Unit Testing

1. Introduction

Marking computer programs in programming courses is, and has always 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 (source code) of the program and its accompanying documentation must be read to evaluate the program structure and style and to ensure that the appropriate standards have been followed.

*Correspondence to: G. Tremblay, Département d’informatique, UQAM, C.P. 8888 Succ. Centre-ville, Montréal, QC, Canada, H3C 3P8, tremblay.guy@uqam.ca

Copyright © 2006 John Wiley & Sons, Ltd.
When dealing with many assignments, marking thus becomes a long, repetitive, and tedious task, which can sometimes make it difficult to ensure consistency and fairness. Furthermore, with the large classes encountered nowadays, marking is often done by teaching assistants (TAs). These assistants may be graduate students or, in some cases, advanced undergraduate students, so the feedback they provide to students may thus be somewhat limited. In addition, the feedback provided to the students 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 that assignment may already be a few weeks away, making the feedback “outdated,” thus less significant. Moreover, if the student made a significant error, it is too late to correct it. Additionally, the lack of early feedback may discourage some students and, in worst cases, might lead them to unethical behavior, i.e., plagiarism.

The typical approach to marking programming assignments thus requires a lot of effort from the instructors or TAs, yet provide little timely feedback to students. Consequently, tools that provide various forms of support for marking programming assignments have been proposed, 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 [1], ranging from script-based execution of test cases [2,3] to metrics-based evaluation [4,5].

In the present paper, we introduce Oto, a tool that provides support for the submission of programming assignments and for their marking. One of its key goal is to reduce the workload associated with the marking task. In addition, it aims at providing timely feedback to the students, as well as feedback before the final submission deadline. Finally, Oto has been designed to be generic and extensible, so that the marking process for a specific assignment can easily be customized and the tool can be extended with various marking components (modules) that allows the tool to deal with various aspects of marking (testing, style, structure, etc.) as well as with programs written in various programming languages.

Some Remarks on our Underlying Pedagogical Approach and Motivation

The initial design of Oto has been influenced by a specific educational context, namely, the teaching of introductory courses in object-oriented programming. 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 specifications (viz., interfaces), not necessarily large programs.
- The students should be introduced early to the key practice of separating the presentation layer and the application logic. In other words, the emphasis should be on developing
classes implementing some appropriate domain model, not on writing detailed I/O routines or developing complex GUIs.\(^1\)

- The teaching of an object-oriented language does not necessarily imply the use of an object-first approach. Thus, the first program that students write is often a simple stream processing program—a program that reads data from stdin and produces output on stdout.
- The important role of testing, both system testing and unit testing, in developing high quality software should be stressed early, as well as the need for appropriate testing framework or tool.

As it will be seen in the following sections, those principles did influence how Oto was designed. However, as our experience in using Oto also shows, many of these principles are also valid for courses that use non-object-oriented programming languages.

Finally, it is important to note that there was also another important, and very practical, motivation that triggered our work on a tool for supporting the marking task at our institution: although some (very) simple Unix scripts had been developed for submitting assignments electronically, those tools were not used in the introductory programming courses, as those courses did not use Unix. Thus, the students were submitting their assignments using diskettes, a practice that clearly created a lot of overhead (mostly for the teaching assistants)\ldots and which might even become soon impossible, as some recent computers come with neither diskette nor CD writer.

Outline of paper

The paper is organized as follows. First, in Section 2, some 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 customizability, genericity and extensibility are achieved is then described in Section 5. Finally, in Section 6, some implementation aspects of Oto are briefly described, followed in Section 7 by our preliminary experience using Oto, and then by concluding remarks together with future work.

2. Existing Tools for Marking Programming Assignments

Preparing a typical assignment in a programming course involves various tasks, as illustrated in Figure 1. A wide variety of tools that provide support for dealing with some of these tasks, generally those in latter phases of the process, have been developed over the years. These tools can be classified into three major categories based on which aspects of the marking process they support:

\(^1\)In the introductory programming courses taught at our institution, graphical user interfaces are indeed introduced, but only in the latter part of the second course.
1. Management of assignments: Various tools support some of the “administrative” tasks that must be handled by the instructors and TAs when preparing and marking an assignment, generally in the later phases of the process: receiving the students’ submissions, keeping track of marks, sending feedback (grade, marking report) to the students after the assignments have been marked, etc. [2, 6].

2. Evaluation of correctness: The main aspect of marking a programming assignment is making sure that the program compiles and works correctly. The usual way to assess the correctness of programs is through testing. Of course, a key goal is to automate as much as possible the testing process, as it can be quite tedious for classes with large number of students.

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” 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 [2].

3. Evaluation of quality: For students to improve their programming ability, it is important to provide them with feedback on the quality of their programs. 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 analyses, based on appropriate design metrics [4, 5]. Quality can be also associated with non-functional run-time behavior, for example, satisfying constraints on the execution time.

Since these three categories, of course, are neither all encompassing, nor mutually exclusive, a given tool can show features from many categories, as shown in the following tool descriptions—see also the paper by Douce, Livingstone and Orwell [1] for an alternative description of marking tools in terms of generations, with a focus on “approaches that automatically assess the success of student attempts to solve programming problems.”

The TRY system [7], introduced in the late 80’s, allowed students to first submit their programs, and then instructors to test the submitted programs. The tests consisted in evaluating the program output based on a purely textual comparison (ignoring possible blank spaces). The instructor could also provide students with output routines, that then had to be used by the students program to generate output.

The ASSYST [2, 3] system allows students to submit their assignments by email. The instructor can then run the system to test and mark the submitted programs, and then send back evaluation reports to the students—thus, feedback provided to students is far from immediate. Marking is done through partially automated testing, based on a context-free grammar specification of the expected output. Other aspects can also be evaluated: quality of the source code (based on McCabe’s complexity metrics or Berry and Meekings’ style metrics); the efficiency of the resulting program (based on the running time or number of instructions executed); effectiveness of the tests developed by the students to test their own programs (based on code coverage).

The BOSS/BOSS2 system [8–10] supports both submission and testing of programs, in various programming languages. BOSS/BOSS2 also provides a form of preliminary checking when an assignment is submitted. Initially, in BOSS, testing was strictly based on the comparison of textual output. In BOSS2, developed for Java, an object-oriented approach was used, where a result produced by a student program can be compared with an expected result, the comparison being done using appropriate methods. BOSS most recent instantiation [10] now uses JUnit for tests. Support for on-line marking and annotations by instructors is also provided, as well as email dispatch of results to students. Plagiarism detection has also been incorporated.

Curator [11] (now called Grader) is a more recent offering that relies on Web technology for submission of assignments. 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 contain any extra lines, or omit any lines” [12]. Furthermore, such an approach requires putting a lot of emphasis on producing program output through console I/O, a secondary aspect...
when using an object-oriented approach to programming that stresses the separation between presentation and application logic.

Another tool based on textual comparison of program output is the OnLine Judge [13,14], a tool which can handle programs in various languages (C, C++, Java). More interesting is that this tool also performs plagiarism detection among the marked assignments, which is done using the sim program [15].

OCETJ [16,17] is a first incarnation of marking tool that we developed with the goal of avoiding reliance on text-based (console I/O) testing. OCETJ, developed in Java, supports both the submission of Java programming assignments by students and the automatic testing of the submitted programs. Testing is done using the JUnit framework [18], a tool that supports the automatic execution of test suites and test cases. Automatic unit testing is a key practice of professional software development [19] which is also becoming prevalent in educational context [20, 21]. OCETJ was also designed with the goal of providing early feedback to the students, through the use of public vs. private test suites—this is explained in more detail in Section 4. OCETJ’s implementation, however, was far from being a generic and portable tool, as it supported only Java programs and JUnit tests.

CourseMarker [22] is a tool which, like our own tool, appears to stress customizability and extensibility, and which can mark programs written in various languages (Java, C++, flowcharts, logic design, etc.). CourseMarker was created as a replacement for Ceilidh [23,24], an older system that was tied to a specific platform and provided no network and graphical interfaces support. CourseMarker incorporates a variety of marking tools—e.g., typographic (layout), testing, specific feature inspection—as well as plagiarism detection. Similar to our own tool, CourseMarker can provide immediate feedback to students and allows the instructor to customize the marking process. Customization is performed using a mark.java customization file that contains standard Java code—this code uses library classes and methods from the com.ltr.cm.marking package to perform the various marking tasks. Testing, contrary to our tool, is done strictly using textual tests, where the “Output is checked by matching to regular expressions that define the desired output” [22, p. 8].

Web-CAT (Web-based Center for Automated Testing) [25, 26] is an assessment tool that emphasizes correctness and completeness of testing... as performed by the students’ programs—i.e., the tool can assess the validity and coverage quality of the tests written by students. Web-CAT can also assess the code style and quality. It provides support for marking Java and C++ programs and the marking process can be customized using XML configuration files.

In the following sections, we describe the Oto tool, designed to be generic and extensible, to support various aspects of programming assignments marking and to provide early feedback to students. Since Oto does not rely solely on testing through textual output but also on unit testing techniques and tools, we introduce these notions in the next section. More specifically, we present the JUnit framework.
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 [27].

As described in the Guide to the SWEBOK [28, p. 5-3]:

Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms, or a larger component made of tightly related units.

As for test automation, which allows for the automatic execution and verification of large number of test cases, it is neither new nor specific to agile methods [29]. 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 performed both early and often [30].

For such an approach to software construction to be feasible, appropriate tools for automating the testing process are required. One well-known such tool is JUnit [18], a unit testing framework for Java.

JUnit has been initially developed and popularized by XP’s proponents (eXtreme Programming) [27]. JUnit embodies a hierarchical approach to the design and coding of test cases: a test class consists of a collection of test suites, where each test suite is composed of a collection of test methods. In turn, a test method—a base test case—is a method whose typical behavior consists in creating some objects, applying some operations on those objects, and then using assertions to ensure that the resulting output or state is the expected one. A key characteristic of such assertions is that they are completely silent if the outcome is the expected one: visible output is generated only when the outcome is not the expected one. Such visible output, when generated, can indicate which assertions failed, together with additional information provided by the tester.

Let us introduce a simple example to illustrate the use of JUnit. Suppose we want to test the class presented in Figure 2, a simple class defining bank accounts. It can be noted 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 appears in Figure 3. The (two) methods whose names start with “test” denote test methods, that is, they 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.

As mentioned earlier, a key feature of assert methods is that they generate no output unless the indicated condition is not satisfied. Whenever this occurs, an AssertionError is thrown. Within the test framework’s context, this exception is then caught and an appropriate message is written to the test log, together with any additional information provided by the tester, for example, a string provided as an optional argument to assertEquals.

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 3. In this example, all methods whose name start with “test” get implicitly included (through reflection) in the test suite by calling the TestSuite constructor with the AccountTest class itself. The resulting test suite can then be executed, as done in method main, and would generate the output presented in Figure 4.
class Account {
  private String cstm;
  private int bal;

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

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

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

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

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

Figure 2. Account class to be tested (with a copy/paste error in method withdraw).

Our example generates textual (console) output through the use of the junit.textui.TestRunner class. Graphical interfaces are also available, for example, by using instead the junit.awtgui.TestRunner or junit.swingui.TestRunner classes.

4. High-level View of Oto

The Oto marking tool, like the OCETJ tool [17] mentioned in Section 2, is designed to support the submission and the marking of assignments through testing based on the use of a unit test framework, and to provide early feedback to the students. Oto has also been designed with two additional goals in mind: to allow instructors to easily customize the marking process they want to use for specific assignments, and to allow the instructors and the tool designers to easily extend the tool to evaluate new and additional aspects. Customizability and extensibility are attained using two mechanisms which are described in Section 5: evaluation scripts and extension modules. Before describing these mechanisms, we present a high-level overview of the use of Oto.

Use Case 1 presents Oto’s high-level use case, in textual form—what Cockburn would call a summary business use case [31]. An equivalent graphical representation appears in Figure 5 using an activity diagram.

To use Oto for a specific assignment, the instructor must first define appropriate evaluations. Generally, two such evaluations would be defined—either of them is optional:

- A public evaluation, used for preliminary verification, thus acting as a filter to ensure that the students’ submissions are minimally correct with respect to the requirements.
import junit.framework.*;
import junit.extensions.*;

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

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

    public void testTransfer() {
        Account acc = new Account( "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 3. AccountTest class for testing the Account class.

There was 1 failure:
1) testTransfer(AccountTest)junit.framework.AssertionFailedError: expected:<100> but was:<200>
   at AccountTest.testTransfer(AccountTest.java:20)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
   at AccountTest.main(AccountTest.java:20)

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

Figure 4. Example of output produced by JUnit for the AccountTest class of Figure 3.
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 define an assignment that will be marked by Oto.

Main Success Scenario :
1. The instructor writes up the assignment and designs the evaluation scripts that will verify and mark the students’ assignments.
2. The instructor creates an assignment submission box.
3. The instructor defines/activates a public evaluation, to provide partial and preliminary feedback to the students.
4. Oto confirms the creation of the evaluation and makes the evaluation script thus defined available to the students.
5. The students check their assignments (to obtain preliminary (partial) feedback).
6. Before the final due date, the students submit their assignments and Oto saves the submitted assignments in the appropriate submission box.
7. The instructor (or teaching assistant) retrieves the assignments from the submission box that were submitted by the students. If no late submission is allowed, the instructor also deletes the assignment submission box.
8. The instructor defines/activates a private evaluation, to mark the submitted assignments.
9. The instructor (or teaching assistant) marks the assignments, using the appropriate, private, evaluation script.
10. The instructor uses the report produced by Oto as an input, along with the students’ hard copy submissions (listing and documentation), to determine the final marks.
11. The instructor deactivates the public and private evaluations.

Use Case 1: Summary use case for Oto.

It is this preliminary verification that provides students with early feedback on their solution.

- A private evaluation, used for the final verification, that is, used to evaluate and mark the submitted assignments.

Before the final submission deadline, students can perform preliminary verifications of their assignments, to receive feedback and ensure that they are “on the right track.” Such submissions are not saved by Oto. When ready, students can then submit their final solutions, in which case the submitted assignments are saved for later marking.

More precisely, the assignments submitted by the students are saved in an appropriate assignment submission box created by the instructor (create_box). Once the final submission date is reached, the instructor (or his surrogate) can perform the final marking of the submitted assignments, as specified by the private evaluation. Each evaluation, whether public
(preliminary feedback) or private (final marking), is defined using an evaluation script, as described in the following section.

5. Customizability and Extensibility: Evaluation Scripts and Extension Modules

The key design goals of Oto were to make the tool generic, customizable as well as extensible. Generic means, first and foremost, not tying the tool to a specific programming language, that is, making it possible to use Oto for marking programming assignments written in various languages. Customizable, on the other hand, 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 performed may vary depending on the programming tool and environment. Furthermore, by a customizable tool, we also mean a tool for which it should be relatively easy, given a library of marking modules, to vary and adapt the overall marking procedure used for any given assignment.

Finally, it should also be possible to extend the marking tool to include additional types of evaluation. For example, the following aspects are or could be handled by marking tools, any of which could be included in Oto by designing an appropriate marking module [32]:

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

In Oto, genericity, customizability and extensibility are attained using two different mechanisms: evaluation scripts and extension modules.

5.1. Evaluation Scripts

An evaluation script—also called an OtoScript—describes the various tasks that need to be accomplished to evaluate a student's assignment. An OtoScript contains a sequence of declarations defining constants, file names, intermediate results, or tasks to be performed—possibly including various computations. Figure 6 presents an example script. (Note: All the examples appearing in the present paper, whether they are Oto scripts or output, have been manually translated to English by the first author—Oto works exclusively en français (in French).)

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 (using the javac module), the other to test the resulting compiled program (using the junit module). The latter task will only be executed if an implicit precondition is satisfied, that is, compilation succeeded without any errors. Explicit preconditions can also be specified using ensure clauses, as shown in the example. Input arguments to a task are specified using keyword parameters, as are the outputs produced by a task, for example, test.nbfailures.

Finally, various numeric results are computed based on the output generated by the test task, and the finalMark thus computed (where a penalty is deducted for each failed test case) is generated.

More precisely, when a student assignment is evaluated using the above script, an evaluation report is produced and contains different elements:
- The explicit output generated by the script (output clause).
- 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.
- The detailed output generated by executing each of the script's task.

As shown in the example script, an identifier can be bound to an expression, indicated by “$(...)”, expression which is evaluated dynamically. Because OtoScripts are evaluated using an underlying Ruby interpreter [33]—see Section 6—, such an expression can contain any valid Ruby expression. In our example, such expressions are used to compute the penalty associated with a failed test as well as the finalMark.
maxMark = 100

assignmentName = Account

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

testClass = Test${assignmentName}

compile :: javac {
    files = $studentPgm
}

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

ensure $($test.nbtests > 0 )
else "Problem: No test succeeded?!"

<<Penalty per error>>
penalty = $( $maxMark / $test.nbtests )

<<Final mark (over $maxMark)>>
finalMark = $( $maxMark - $penalty*$test.nbfailures )

output { finalMark }

Figure 6. An example evaluation script.

Figure 7 presents an example report for the script of Figure 6, in the case of a program whose execution fails for one execution out of two.†

An evaluation script is always defined with respect to a single assignment. In other words, a given script can be used indifferently for performing a single preliminary verification (check_assignment using a public evaluation), for marking a single student assignment (check_assignment, but generally using a private evaluation), or for marking a whole group of assignments (mark_assignments, generally using a private evaluation).

†Again, this example has been translated from French. Furthermore, output from the Junit execution has been partly omitted, as indicated by “[..]”. Also note that Errors in the JUnit output must be distinguished from Failures: a failure is when an assertion is false, whereas an error is when an exception occurred but was not caught.
5.2. Extension Modules

Evaluation scripts can be used to create evaluations for specific assignments, using existing tasks. However, scripts cannot be used to introduce new mechanisms to evaluate assignments, i.e., they cannot define new types of evaluation tasks. Instead, such new tasks are defined by designing extension modules.

Here are some examples of extension modules which have been or could be developed:

- Modules that interface with various compilers, thus allowing Oto to process assignments written in different programming languages.
• Modules that interface with various test frameworks, again allowing Oto to test programs written in various languages.

• Modules that treat various quality aspects of a student program, such as those mentioned earlier, for example: Does the program obey the required programming/documentation style guidelines or standards? Is the running time acceptable? Does the program appear to be original work by the student relative to the other submitted assignment (plagiarism detection)?

• If students are required to submit their own tests, module that evaluate those tests for various forms of coverage [34].

Contrary to evaluation scripts, which can readily be developed by instructors using the OtoScript notation, extension modules will generally be developed by the Oto tool designers and developers. An extension module is a component that provides a well defined interface, with a number of input arguments and a number of output results. Documentation describing the interface of an extension module can be obtained using the Oto `describe module` command.

Extension modules essentially play a role of proxy between Oto and existing tools. For example, the `javac` and `junit` extension modules package the information from the script, run the associated program, then interpret the results generated by running the external program to make them readily available for use in the script report through the task output results.

Extension modules have been developed using the Ruby scripting language [33] and must satisfy three conditions. First, the name of the Ruby class defining the extension module must obey an appropriate naming convention—`oto_ext_name_of_task`. Second, it must implement a `run` method which takes three parameters: a list of arguments specified in 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 module. 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. Existing Scripts and Modules

To use Oto and define their own evaluations, instructors have to learn to write OtoScripts. To facilitate this task, various skeleton scripts are provided to instructors, which should make it relatively straightforward for them to develop their own scripts.

Of course, various extension modules are also provided. The following extension modules are now available—again, for the English reader’s benefit, module names as well as description of their interface have been translated from French:

`javac`: A module that compiles Java programs. The module’s interface consists of the following items:

- Inputs: `sourceFiles`, together with optional inputs—`classpath`, `compilingOptions`, `root`.
- Output: `stdout` (detailed output generated by the Java compiler).

Note that when compilation fails, the evaluation script is always aborted.
junit: A module that tests compiled Java programs using the JUnit test framework. The module’s interface consists of the following items:

- Input: testedClass, together with optional inputs—classpath, compilingOptions, root.

test_filter: A module that performs system-level tests for filter programs based on textual comparison—a filter program is one which exclusively reads input on stdin and generates output on stdout; arguments may also be transmitted, when initially calling the program, through the argv array of strings. The various tests cases are defined with at least two text files and, optionally, with a third one. For example, for a test case named X, a first file (testX.data) would describe the inputs to the program, and a second file (testX.results) would describe the expected (textual) output. The comparison between the expected output and the program’s output is then performed using Unix’s diff command. An additional file (TestX.arguments) could also be used to provide arguments when calling the program.

The module’s interface consists of the following items:

- Inputs: filter (the executable program to be tested), testDirectory (the directory containing the test*.{data,results,arguments} files for the various test cases), along with an additional optional input (root).

gcc: A module that compiles C programs. The module’s interface consists of the following items:

- Inputs: sourceFiles, together with optional inputs—compilingOptions, root.
- Output: stdout (detailed output generated by the C compiler).

Like for the javac module, when compilation of the C program fails, the evaluation script is aborted.

cmd_bash: The most general of the extension modules, since it can execute an arbitrary shell command or bash script—including a call to make, thus allowing the execution of an arbitrary sequence of operations.

The module’s interface consists of the following items:

- Inputs: command or script (to be executed with bash), summary, item1, along with additional optional inputs (root, item2, item3, item4).
- Outputs: result1 (and, possibly, result2, result3 and result4), nbErrors, detailedOutput.

The output produced by the command or script execution must contain a summary line, used to generate the various resulti outputs. This special line of output is signaled by
<<Tests for a first run>>
*run1:: cmd_bash
{
  command = "make -s run1"
  summary = "Result summary:"
  item1  = "executed tests = NUMBER"
  item2  = "failed tests = NUMBER"
}
<<Number of executed tests>>
nbTests = $( $run1.result1 )

<<Number of failed tests>>
nbFailures = $( $run1.result2 )

<<Success rate>>
successRate = $( 100.0 * (nbTests - nbFailures) / (1.0 * nbTests) )
output { nbTests, nbFailures, successRate }

Figure 8. An example script that uses the cmd_bash extension module.

the string summary. The different fields of the summary, which must include a numeric result, are then specified using the item1 field (and, optionally, item2-item4), where the appearance of a numeric result is indicated using the special keyword NUMBER.

The use of summary and the various item fields makes it possible to parameterize an evaluation script based on how a shell command or script generates its output summary. For example, suppose a test program whose execution is defined through a makefile generates the following output summary:

Result summary: executed tests = 10; failed tests = 2;

Figure 8 presents an appropriate OtoScript illustrating the use of the cmd_bash extension module to call this test program and use its resulting output.

6. Oto’s Design and Implementation

In this section, we first present the general design and implementation of the Oto marking tool. We also briefly describe how Oto is physically deployed, thus describing the concrete interfaces through which Oto can be used. Finally, we mention some of the key issues and difficulties that had to be handled while implementing Oto.

6.1. Oto’s High-level Design

Figure 9 presents the overall software (static) architecture of Oto, which consists of various files and modules, among which we briefly describe the following:
Figure 9. Oto’s High-level (static) architecture.

Figure 10. Overview of data flow among Oto’s components.
• **Commands**: Each Oto command is implemented by a specific Ruby class, defined in a file within the `Commands` directory, and whose execution is performed by a `run` method. For example, the `mark_assignments` command is implemented in class `MarkAssignments` and defined in file `otocmd_mark_assignments`. The `run` method of this class, when passed an appropriate `Execution context`, will perform the various tasks associated with the marking of a whole group of assignments. The loading of the appropriate class is then made on demand, at run-time, by the command dispatcher.

• **Extensions**: This is similar to **Commands**, but for evaluation tasks defined by extension modules. For example, the `test_filter` task is defined by an `OtoExtTestFilter` class in file `oto_ext_test_filter`. Again, loading the appropriate class is made dynamically when an evaluation script is executed and the indicated task is encountered.

• **Script translator and Interpretation engine**: When an evaluation is created, whether private or public, an appropriate Oto script must be specified. At that point, the script is translated into an intermediate form. When the evaluation script is executed—to perform some preliminary verification of an assignment in the case of a public script, or to perform the final marking in the case of a private script—that intermediate code will then be interpreted by the `Interpretation engine`.

Figure 10 presents a graphical overview of the data flow among Oto’s key components.

### 6.2. Some Implementation Aspects

Every instructor has a dedicated directory managed by Oto—called his Oto space—and located in his home directory on the Unix machine on which Oto runs—currently, an Ultra-Enterprise Sun machine running Solaris 5.8. This Oto space is used for storing the assignments submitted by the students as well as storing the evaluations created by the instructor—recall that an evaluation is defined through the combination of an Oto script, generally along with auxiliary files (e.g., tests cases or tests programs). To ensure security and privacy, the various Oto commands are run as a **SGID** (Set Group Id) Oto program. Therefore, only the Oto user and the instructor may access the files created in the instructor’s dedicated directory. Similarly, when a student executes some Oto commands (mostly, `check_assignment`), a working Oto directory is also created in his home directory.

Execution of a script occurs either locally in the student’s directory (preliminary verification using the `check_assignment` command) or in the instructor’s directory (final marking of a group of assignments using the `mark_assignments` command). Thus, when a student uses an instructor’s script (defined through a public evaluation) to assess his own assignment, the script is run using the student’s identity and rights in the directory where the command was launched, i.e., in the student’s directory, so that any possible side effects remain under the student’s responsibility. Similarly, when the instructor marks a group of assignments, execution is performed within his own account. Again, and this would also be the case for assignments submitted through some other media (e.g., diskette or CD), possible side effects are under the instructor’s responsibility. Thus, it is generally advisable that special accounts, preferably with limited privileges and quotas, be used for retrieving and marking groups of assignments.
METHOD CmdOtoCheckAssignment.run( argv )
BEGIN
  instructor ← argv₁
  eval ← argv₂
  submittedFiles ← [argv₃, ..., argv[argv_length]]
  Copy all submittedFiles in a temporary working directory
  From instructor’s Oto space, retrieve the (pre-compiled) evaluation script named eval
  and copy it in the temporary working directory
  Interpret the (pre-compiled) evaluation script eval from within the working directory
  Clean-up the working directory
  Return the results generated by the script’s evaluation
END

Figure 11. High-level algorithm for the marking process.

The high-level algorithm for the marking process is informally described in Figure 11. More precisely, this algorithm describes how an assignment for a specific student is verified. Thus, it describes what would occur when executing a command such as the following, where tremblay is the instructor’s name, problem1 the evaluation name, and foo.java the only submittedFiles (i.e., |argv|=3):

    oto check_assignment tremblay problem1 p1.java

When reading such a command, the main program performs a dispatch to the appropriate method using the command design pattern [35]. More precisely, executing some_command is done by packing the command-line arguments, creating an instance of the CmdOtoSomeCommand class and then executing its associated run method.

For a student to be able to perform a preliminary check of his solution, the instructor must have previously activated an appropriate evaluation, in this case, using a command such as the following:

    activate_eval --public problem1 p1-script.oto

When the instructor publishes such an evaluation script, the script is immediately checked for syntax errors and then pre-compiled, i.e., an intermediate representation that uses a simple and uniform instruction format is created and stored in the instructor’s Oto space for subsequent use.

When interpretation of an evaluation script is required, as it is the case in Figure 11 (Interpret...), the pre-compiled intermediate code is loaded and interpretation begins. The Interpretation engine (Fig. 10) then reads each line/command of the (pre-compiled)

---

5 Publishing an evaluation script does not mean the associated tests are also made public. If the instructor wishes to make the tests public, he must do so explicitly, not through Oto.
evaluation script and evaluates it in the context of the temporary directory created by the invoking command. When a task defined by an extension module is encountered—such as javac or junit—the interpretation engine then loads and runs the appropriate Ruby (proxy) module. Such a module can be one of the predefined modules (described in Section 5.2) or could be a local one provided by the instructor. Whenever an error occurs during script interpretation, the error is caught and reported (last line of Fig. 11). Of course, when marking a group of student assignments, Oto tries to evaluate them all, without aborting the whole job because of a single bad assignment.

6.3. Concrete User Interfaces and Physical Deployment

Oto’s core has been implemented as a Unix tool. Thus, its basic interface is through the command line, with a single entry point but with many subcommands (à la cvs [36]). The general form of an Oto command is then the following:

\[
\text{oto oto-command [cmd-option] [cmd-args]}
\]

Table I briefly presents, more or less in temporal order of use, Oto’s key commands. The first column, with I or S, indicates whether the command is intended for Instructors or Students. Additional commands are also available but have been omitted, e.g., describe_module, destroy_space, list_commands, suppress_multiple_submissions, etc.

Two additional interfaces with Oto have also been developed:

- In the first two introductory Java programming courses taught at our university, the students use the BlueJ environment [37]. Thus, we developed a BlueJ plug-in that allows
students to interact with Oto directly from within the BlueJ environment, whether or not BlueJ is running on the Oto Unix server.

Using this plug-in simply requires that it be installed in the proper BlueJ/lib/extensions directory on the same local machine as BlueJ. An appropriate ssh tool must also be installed, since a key restriction is that a student must necessarily have a user account on the Oto Unix server: password and security issues are addressed through the establishment of an appropriate ssh session, i.e., are handled neither directly by the plug-in nor by Oto but by the Unix server.\footnote{At our institution, this is not a major restriction, since all students have an account created for them on the Unix server as soon as they are registered in any CS course.}

Since this plug-in is used solely by students, it supports only the commands directly intended for students, for example: list_evaluations, verify_assignment, list_boxes, submit_assignment and confirm_submission.

- Although interesting for the introductory Java courses, the BlueJ plug-in cannot be used in more advanced courses. Thus, we also developed a more general Web interface \cite{38}.\footnote{http://labunix.uqam.ca:8181/~oto/application-web}

This Web interface, in fact, consists in two different sets of Web pages:

- A set of Web pages used by instructors to create submission boxes and evaluations, retrieve and mark assignments, delete boxes and evaluations, etc.
- A set of Web pages used by students to list available boxes and (public) evaluations, to perform preliminary verifications of their assignments, to submit

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I create_box</td>
<td>Create an assignment submission box.</td>
</tr>
<tr>
<td>I activate_eval</td>
<td>Activate an evaluation, private (default) or public (--public).</td>
</tr>
<tr>
<td>S list_evaluations</td>
<td>List the (public) evaluations defined by an instructor.</td>
</tr>
<tr>
<td>S check_assignment</td>
<td>Perform a preliminary verification of an assignment</td>
</tr>
<tr>
<td>S list_boxes</td>
<td>List the assignment submissions boxes created by an instructor.</td>
</tr>
<tr>
<td>S submit_assignment</td>
<td>Submit an assignment into the indicated box.</td>
</tr>
<tr>
<td>S confirm_submission</td>
<td>List the assignments submitted for a specific instructor and submission box.</td>
</tr>
<tr>
<td>I retrieve_assignments</td>
<td>Bring, in the current directory, all the assignments submitted by students in the indicated box.</td>
</tr>
<tr>
<td>I destroy_box</td>
<td>Delete a submission box.</td>
</tr>
<tr>
<td>I deactivate_eval</td>
<td>Deactivate an evaluation.</td>
</tr>
<tr>
<td>I mark_assignments</td>
<td>Mark all the indicated assignments.</td>
</tr>
</tbody>
</table>

Table I. List of Oto’s key commands.
their assignments and then confirm their submission, etc.—in other words, the same commands as those provided by the BlueJ plug-in.

Both the instructor’s and the student’s Web interfaces use the same mechanism as the BlueJ plug-in for enforcing security constraints, that is, the establishment of a proper ssh session using the student’s or instructor’s Unix username and password.

6.4. Key Implementation Issues

Some of the key implementation issues that had to be addressed while developing Oto were the following:

- Data confidentiality and integrity: Ensuring the confidentiality and integrity of the assignments submitted by students as well as the private evaluations created by instructors was, most definitely, one of the key implementation issues. Whether for a preliminary verification or for a final submission, an assignment submitted by a student should not be accessible by other students. For a preliminary verification, this is ensured simply by evaluating the student’s assignment within his own space. For a final submission, the situation is different, since the student’s files need to later be made available to the instructor—when he uses the retrieve assignments command. The submission box (i.e., directory) created by the instructor is accessible only to the instructor and to users whose group is oto, a group which contains a unique user, namely, Oto itself. When a student submits an assignment, the files are transfered indirectly into the instructor’s submission box. During the first phase, the files are transfered into some private space, a task done, in part, without any special privileges. In the subsequent phase, the files are transfered into the instructor’s space, using Oto’s privileges. The resulting files are then owned by the student, with access permissions to Oto. Later on, when the instructor executes the retrieve assignments, ownership is transfered to the instructor, but now with exclusive access rights to the instructor. For stronger security, the Oto commands, as alluded earlier, are run as a SGID (Set Group Id) Oto program, not as SUID programs. Running Oto as such a program required that Oto’s underlying Ruby program be ran in SAFE mode, thus requiring that all external data be properly cleaned up (untainted) to ensure proper execution. Finally, regarding integrity, Oto ensures that all submission operations are atomic. Whenever a (non-recoverable) file transfer error occurs, the files already transfered are deleted, thus restoring the target directory to its initial state.

- Dynamic loading of extension modules: As described earlier, a small collection of predefined extension modules is available to instructors. However, we also wanted to make it possible for an instructor to define his own extension modules, and use them in his evaluation scripts, thus the need for dynamic loading of modules (at script execution time). Oto has been implemented using the Ruby programming and scripting language [33]. Using such a language, together with an appropriate naming convention (see Section 6.1), made it relatively simple to locate and load code dynamically, on demand. Thus, Oto’s extensibility is, in fact, dynamic.
• Error recovery and handling: Oto is a multi-layer software and, as is generally the case with such software, recovering from errors and producing meaningful messages is sometimes quite challenging. With Oto, errors may be associated with invalid user inputs (arguments to the various commands), file errors (e.g., invalid access rights for a submitted file), syntactic or interpretation errors in evaluation scripts, errors in student programs, etc. Furthermore, it is vital that errors in student programs do not abort script execution. To better circumscribe the location and meaning of errors and support error handling, we develop a set of assertion and error signaling operations which dynamically generate appropriate error classes. Such an approach was possible because of Ruby’s overall dynamic behavior.

• Unit and system testing: Oto comes with a full set of automated unit and system tests: The source code is over 11 KLOC, and, as typical in such projects, more than 40 % is test code. For the tests, we used the well-known Ruby Test/Unit framework**. In Test/Unit as in many similar test frameworks, the unit tests are independent one from another. For some critical tests, however, we encountered the need for stating explicitly the (generally, hierarchical) dependencies that exist between various tests, so that if a test fails, its depending tests do not execute. Thus, we also designed our own test extensions that allow a class to explicitly state the required dependencies.

Further details about the implementation of Oto can be found in the second author’s thesis [39].

7. Preliminary Results

At the time of writing, Oto’s full capabilities have been used with success in various courses:

• In a second year “Construction and maintenance” course, using C.
• In a third year “Parallel programming” course, using the MPD programming language [40].
• In a first year “Introduction to programming I” course, using Java.

In addition, Oto’s support for electronic submission and retrieval of assignments has also been used in other courses.

Construction and maintenance course In this course, the first time Oto was used, the students had to do three assignments. In one section of the course, all three assignments were submitted using Oto, although only the first and third assignments were marked using Oto, whereas in a second section, only the third assignment was handled using Oto—in both cases, no preliminary (public) evaluation was made available.

More precisely, for the first assignment, the students had to write a program that read an ASCII image file (pgm format) from stdin and that generated on stdout an inflated version of the image (through a simple interpolation). The students’ programs could thus be tested

**http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc
based on textual output comparison, i.e., using an evaluation script that relied on the gcc and test_filter extension modules. Thirty assignments were marked and everything ran smoothly.

Marking the third assignment was a bit more complex. This assignment was a maintenance assignment, in which the students had to modify some existing software to improve one of its functionalities [41]. This software, a C program for managing a collection of books, consisted of 10 modules (over 25 different files, counting the various test programs) and came with both automated unit tests (à la JUnit, i.e., assertion-based) and system level tests (based on test cases described using text files). The modified program had to satisfy the same set of system level tests. However, additional unit tests had to be defined by the students, so that the program as whole also had to satisfy a larger set of unit tests. This assignment was marked using an evaluation script that relied on an earlier version of the cmd_bash extension module—more precisely, a module now rendered obsolete by the more general cmd_bash new version. The evaluation script job was to run both the automated system level tests and the unit tests. Various marks were then computed by the script, namely, the success rates for the system level tests and unit level tests, along with the number of new additional test case and assertions—an approximate measure of the effort students put into defining their own tests.

For this third assignment, 55 student assignments were processed by Oto. Although a few (namely, four) assignments could not be properly handled, the problem was not because of Oto. Rather, it turned out that some of the files submitted to Oto were not correct. More precisely, all the files for the software on which the students worked were under the control of CVS. For submitting their final solution, the students had to use a (shell) script that created a patch, generated by a call to cvs diff, and submitted the resulting patch to Oto. That patch was later processed by the patch utility, from within the makefile used by the evaluation script. In the case of the four problematic assignments, for reasons beyond the scope of the current paper, the submitted patch turned out to be empty. Hopefully, in anticipation of such problems, an archive for the whole set of files was also created and submitted to Oto by the submission script, so the students’ solutions could be recovered.

Since then, Oto has been used regularly in this course, for various assignments, with full success.

Parallel programming course In this course (with a single section), the students also had to do three assignments. All three assignments were submitted and marked using Oto—more precisely, the third assignment consisted in three different independent exercises, two of which were handled by Oto. No preliminary evaluation script were defined, as the test cases and test scripts (including appropriate makefiles) were all distributed directly to the students.

The use of Oto in this parallel programming course was quite helpful in the following way: all programs submitted by students were parallel programs, so that many different runs of each program had to be performed to ensure, as much as is possible through testing, that the program did not contain any race condition. So, for instance, each submitted program was executed using one, then a small number, and then many threads and processors. Furthermore, obtaining some parallel acceleration was part of the assignments’ requirements, which required measuring the execution time, again with varying number of processors. All such runs and computation (for the running time and acceleration) were performed using Oto scripting facilities.
Introduction to programming course  Our introductory programming course is an “imperative-first implementation” CS1 course [42] using Java. Thus, students are first introduced to the basic notions of imperative programming, whereas objects and classes are introduced only later in the semester, and studied in more detail in the subsequent (CS2) course.

In this course, tests were performed to use Oto for processing students’ submissions for weekly (two hour) closed laboratories, of which four (out of 14) are counted in the final course grade. Students were first introduced to the use of Oto, using the Web-based interface. Then, during the following weeks, the marked laboratories were interspersed with non-marked ones.

Because of the imperative-first approach, the progression of the laboratories was along the following lines:

- Filter programs: The first laboratories dealt with main programs that performed textual input/output on the standard streams (i.e., filter programs), thus dealt mostly with constants, variables and control structures—students did not have to deal with the details of input processing, as they were provided with a simple Keyboard class.
- Programs providing library of (class) methods: The following laboratories dealt with procedural abstraction, both for scalars and arrays. Since the notions of class, object and, thus, instance method had not yet been introduced, students had to develop methods which were public and static.
- Simple classes: The later laboratories dealt with basic data abstraction. Students had to define and manipulate simple classes and objects.

Experience using Oto’s basic commands showed us that one could define evaluations and tests suitable for these three types of laboratories. However, doing so required multiple steps, and defining the appropriate tests cases, whether they were textual test cases (filter programs) or JUnit tests (methods and simple classes), was rather long and tedious. Furthermore, the complexity of defining such tests was quite beyond the scope of novice students’ knowledge. Thus, although students could use and apply the tests provided by their instructors, they could not define their own tests. Thus, this lead us to extend Oto with new commands that simplify the instructor’s task, also providing testing support to students [43].

Having novice students use a tool such as Oto early in their first course also lead us to identify some possible improvements. For instance, during the first week, we soon realized that students, even when given specific instructions about which file to submit in which submission box, often… did not follow the instructions. The Oto command used to create a submission box was thus extended with options allowing an instructor to specify various constraints, for example, the submission box accepts a unique file with a specific name, or the submission box accepts multiple files, among which one with a specific name must be present. In fact, it is now possible to associate any evaluation script, such as those described in Section 5.1, with a submission box. For example, it would be possible to configure a submission box so that only programs which correctly compile and run some minimal tests are accepted—of course, this would be a bit stringent in an introductory course.

Other options were also added. For example, in its original instantiation, Oto did not implement any mechanism to automatically disable a submission box once a certain deadline occur. It was simply assumed that the instructor would, when appropriate, disable the submission box. For closed laboratories with a fixed and relatively short duration (two hours,
OTO, A GENERIC AND EXTENSIBLE TOOL FOR MARKING

with a leniency delay of approximately 15 minutes), this was not appropriate, as the instructor was not necessarily available at the exact time when the submission box had to be deactivated. Thus, a deadline specification option was added to the submission box creation operation.

Another improvement was related to handling of “erroneous programs.” The first laboratory, which dealt with loops, showed us that a typical error of novice programs was not handled properly, that is, programs containing an infinite loop . . . of output. For programs performing no textual input/output, erroneous programs containing infinite loops were properly handled through a time-out mechanism—when marking a program, one can specify the maximum CPU time the program is allowed to use. However, for erroneous programs looping infinitely and performing textual output, the time-out turned out to be insufficient . . . since a file space overflow could occur before the execution delay expired. To deal properly with such programs—which, as we found out, occur regularly in every laboratory, including simple ones which required no loop (sic)—an option was added to specify the maximum size of output that a program is allowed to generate—more precisely, this option specifies the maximum file size, textual output being associated with a file through stream redirection.

8. 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 Oto. 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.

Oto’s key characteristics are the following:

- Oto is a marking tool which is generic and customizable, allowing the instructors to customize his marking process using marking script.
- Oto is also an extensible marking tool, since additional extension modules can be “easily” developed. For example, the extension modules gcc and cmd_bash used for marking some of the course assignments described in Section 7 have both been developed in about an hour by the first author.

In the short term, we are planning to start using Oto in our first year programming course in Java. As mentioned earlier (Section 7), this course has a weekly hands-on laboratory, some of which (4 out of 14) have to be handed-in and are then marked by the teaching assistants. Tests performed in that context allowed us to improve Oto, which is ready “to be put in production,” for use in both the laboratory assignments and the regular (larger) programming assignments.

Easily for the Oto developer, not necessarily for an instructor, since developing a new extension module requires some basic knowledge of Ruby and Oto’s underlying architecture. However, our experience so far in developing such new modules shows that it is rather straightforward.
Exploring further Oto’s extensibility is at the core of our future work. Thus, we are planning to extend Oto by defining additional extension modules, which will allow Oto to treat various marking aspects beyond testing. For instance, we plan on developing an extension module that would perform various stylistic checks, for example, using Checkstyle\(^\text{\dagger}\). We would also like to develop an extension module that would compute various quality analysis and metrics, with the aim of providing further input to help the instructor or the teaching assistant to determine the students’ final grades. 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 a plan for an extension module that would detect plagiarism among the submitted assignments, a plague with which, sadly, most universities are now confronted.

Additional information, in French, about Oto—including source code, on-line documentation, a user manual [44], a student’s guide for using the Web interface [45] and an instructor’s guide for developing laboratories and assignments [46]—can be obtained from the following Web site: http://www.info2.uqam.ca/~oto.

ACKNOWLEDGEMENTS

First of foremost, we acknowledge the financial support provided by UQAM (Université du Québec à Montréal) through grants from the Fonds de développement pédagogique (2003–04 and 2005–06) and from a Projet d’intégration des chargés de cours (2003–04).

Special thanks to Matthieu Fecteau and Éric Bélanger, both undergraduate students, who developed the BlueJ plug-in, to Bruno Malenfant, a lecturer in our Departement, who developed the Oto script translator, and to Mohamed Takim, a graduate student, who developed the Web interface. Thanks also to Alpha Boubacar Diallo, Louise Laforest, Bruno Malenfant et Emmanuel Chieze who were among the first users of Oto.

REFERENCES


\(^\text{\dagger}\)http://checkstyle.sourceforge.net
OTO, A GENERIC AND EXTENSIBLE TOOL FOR MARKING


