Semi-Automatic Marking of Java Programs Using JUnit

Guy Tremblay
Dépt. d'informatique, UQAM
C.P. 8888, Succ. Centre-ville
Montréal, QC, H3C 3P8, Canada

Éric Labonté
Dépt. d'info., Cégep du Vieux-Montréal
255, rue Ontario Est
Montréal, QC, H2X 1X6, Canada

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 (structure and style), etc. With the large classes encountered nowadays, the feedback provided to students through marking is thus generally rather limited, and often late and outdated.

Tools providing support for marking programming assignments do exist, ranging from support for administrative aspects (submission and management) through automation of program testing or support for source code evaluation based on metrics. Tools that support program testing generally rely on comparison of *textual* output, requiring students to spend much effort dealing with details of input/output.

In this paper, we introduce a tool that provides support for submission and testing 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, testing of the programs *does not* rely on testing program textual output but instead on the use of an appropriate testing framework (viz., JUnit). Such a framework allows testing to be done on the key features of the software unit, that is, the underlying domain model rather than some (textual) presentation.

Keywords: Technology in Education, Automatic Marking, Introductory Programming, 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 in most universities and colleges, marking is generally done with the help of teaching assistants. These assistants may be graduate students or, in certain cases, advanced undergraduate students. The feedback they provide to the students may thus be somewhat limited and, more importantly, may come too late in the learning process of the students, as the whole marking process for classes with a large number of students is rather lengthy. For instance, when a student finally receives her graded assignment, the topic dealt by this assignment is already "outdated", so less significant for the student. Furthermore, if the student made an important mistake, it's too late to make any correction. In other words, the typical approach to marking programming assignments requires a lot of effort from the instructors and/or teaching assistants, 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 [11, 18].

In the present paper, we introduce a tool, called OCETJ [16], that provides support for the submission of programming assignments as well as for their testing. 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.

Underlying pedagogical approach

The design of OCETJ has been influenced by the specific educational context, namely, the teaching of *introductory* courses in object-based programming. [1] We claim that some of the pedagogical principles underlying such courses should be 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

[1] We say "object-based", as opposed to "object-oriented", because although the notions of object, class, even interface, are introduced early in the course, inheritance *is not*.

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 graphical user interfaces. This does not mean, however, that appropriate *views* should not be available. Instead, these views should be provided *by the instructor*, allowing the resulting programs to go beyond the IPO style (Input/Process/Output) generally associated with plain, textual, console I/O, as is often the case in introductory courses.

- The important role of unit testing in developing quality software should be stressed. For instance, the use of a test-first approach, as recently promoted by proponents of eXtreme Programming [2, 4], should be introduced early. As it will be shown in Section 3, using an appropriate unit-testing framework supports such an approach and, furthermore, alleviates the "what should the `main` program be?" problem often associated with an early introduction of objects.

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, the popular unit test framework developed for Java, are presented. The next section then presents OCETJ, our own tool, followed by a brief conclusion.

2. 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 major features of these tools can be classified into three categories:

1. Management of assignments: Tools in this category support the various administrative tasks that must be handled by the instructors and teaching assistants: 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, of course, through testing. A key goal should be to automate as much as possible the testing process, which can become quite tedious given a large number of programs to test.

   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 [11, 18].

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 [21] 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 [19, 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* (not to say *nitty-picky*). For instance, the student documentation for Curator clearly indicates that "It is important that your output file not contains any extra lines or omit any line."

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. As it will be shown in more detail in Section 3, unit testing, especially in the context of an object-based approach to programming, does need not to (nor should it) rely on producing traces or textual program output.

The OCETJ tool described in Section 4 supports both the submission of assignments by the students and their automatic testing, but not (yet) the evaluation of their quality. It aims also at providing early feedback to students by allowing them to perform *tentative* (i.e., non-final) submissions in order to know whether their programs appear to work correctly on a subset of the test cases. More importantly, and this is its key characteristic, it precludes testing through textual output. In order to better understand this feature, we first introduce the JUnit framework for unit testing.

3. THE JUNIT TESTING FRAMEWORK

Although the importance of testing in developing correct programs has always been recognized, proponents of agile methods — among which *eXtreme Programming* (XP) is the most well-known approach — have recently emphasized the beneficial role of unit testing and test automation [2, 20]. For instance, XP's proponents advocate the use of a *test-first* approach (also called "*Test-Driven Development*" [4]), i.e., the discipline of writing automated test cases *before* writing the code, summarized in the following *motto*: "Never write a line of functional code without a broken test case" [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 XP and agile methods is the tight integration of test automation with a test-first approach to code development. This allows testing to be done both early and often. Thus, frequent regression testing can be performed, ensuring that the software works correctly whenever changes are made, whether these changes result from the addition of new code to handle additional features, the modification of existing code to fix bugs, or from refactoring done on the existing code to improve its quality and maintainability [9, 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, which we briefly explain in the following paragraphs.

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); for simplicity, also note that amounts are simply `int`egers. A JUnit class for testing the key methods of the `Account` class (for simplicity, exceptions are ignored) is presented in Figure 2. The `setUp` method is used to initialize a number of global variables, which can subsequently be used by the various test cases. Each of the following methods, whose names all start with "`test`", then represents a specific *test case*. These various test cases generally use `assertEquals` to check whether the result returned by the method being tested (second argument of `assertEquals`) is the

```
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`)

expected one (first argument). Other variants of `assert` methods do exist, for example, `assertTrue`, `assertNotNull`.

A key and common feature of all these `assert` methods is that they generate no output *unless* the expected condition is *not* satisfied. Whenever this occurs, an `AssertionFailedError` is thrown. In the context of the test framework, this exception is then caught and an appropriate error message is written to the test log. [2]

A collection of test cases is called a *test suite*. Each test class must define its associated test suite by implementing a `suite()` method. This can be done by explicitly allocating a `TestSuite` object and then adding the various test cases as done in the following `suite()` method:

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new AccountTest("testGetCustomer"));
  suite.addTest(new AccountTest("testNewAccount"));
  suite.addTest(new AccountTest("testTransfer"));
  return( suite );
}
```

A much simpler approach can be used, however, as illustrated by method `suite()` in Figure 2. In this case, the `TestSuite` constructor is called with an argument indicating the name of the test class. All methods from that class whose name start with "`test`" then get included in the test suite, which is made possible through Java's *reflection* mechanism. This method of generating a test suite from a collection of test cases explains why the names of the various test cases are usually chosen to be of the form "`testSomeFeature`".

[2] This exception can also be caught by the code within the test case, for example, to generate a more specific and detailed error message, which is done by making a call to the `fail` method.

```
public class AccountTest extends TestCase {
  private Customer c1, c2;
  private Account acc1;

  public AccountTest( String name ) {
    super(name);
  }

  protected void setUp() {
    c1 = new Customer( "Tremblay" );
    c2 = new Customer( "Labonte" );
    acc1 = new Account( c1, 100 );
  }

  public void testGetCustomer() {
    assertEquals( c1, acc1.customer() );
  }

  public void testNewAccount() {
    Account acc = new Account( c2, 200 );
    assertTrue( acc.customer() == c2 &&
                acc.balance()  == 200 );
  }

  public void testTransfer() {
    int initBal = acc1.balance();
    acc1.deposit ( 50 );
    acc1.withdraw( 50 );
    assertEquals( initBal, acc1.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`

The resulting test suite can be executed as shown in method `main` of Figure 2. Whenever this test program is run, each of the test cases in the associated test suite (all methods starting with "`test`") will then be executed (in arbitrary order). In this example, the use of the `textui` variant of `TestRunner` indicates the use of a command line interface, so that output similar to the following would be produced:

```
There was 1 failure:
1) testTransfer(AccountTest)
    junit.framework.AssertionFailedError:
    expected:<100> but was:<200>
at AccountTest.testTransfer(AccountTest.java:31)
at AccountTest.main(AccountTest.java:39)

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0
```

Graphical interface test runners (e.g., `swingui`, `awtui`) are also available.

The use of unit test frameworks — variants do exist for other programming languages, see `www.xprogramming.com` — and the integration of a disciplined test practice into introductory programming courses will probably become more prevalent in the coming years. The use of JUnit in an introductory course at the university level has already been reported [1] and the second author of the present paper started experimenting with the use of JUnit in a second year Java course (within a technical college degree) in the fall of 2002. Our goal in the present paper, however, is not to discuss the use of a testing framework in an introductory programming course *per se*. Rather, our goal is to show how this tool can be used to help automate the testing process associated with the marking of large number of programming assignments, which we explain in more detail in the next section.

Before proceeding further, however, let us note that our pedagogical approach (programming-in-the-small, little emphasis on presentation and I/O) combined with the use of the JUnit framework implies that the code submitted by students will contain various classes implementation, but no main program. More precisely, one possible main program will be defined by the test classes defined by the instructor. Of course, other main programs can also be made available to the students, that is, different *views* of the same domain model can be provided, for example, a command line view or a simple graphical user interface.

4. OCETJ: A TOOL FOR SEMI-AUTOMATIC MARKING OF JAVA PROGRAMS

The OCETJ tool (*Outil de Correction et d'Évaluation de Travaux Java* = marking and evaluation tool for Java assignments) was developed with two major goals in mind: i) provide support for instructors (and teaching assistants) for marking programming assignments; ii) provide early feedback to students. In this section, we first describe the core functionalities of OCETJ, followed by a brief overview of its implementation.

Core functionalities of OCETJ

A use case diagram [17] identifying the primary actors (users) of OCETJ along with the key use cases is presented in Figure 3. Three types of actors use the system: *Instructors*, *Students*, and *Teaching Assistants*.

In order to enforce proper security, each actor must first *Register* into the system before he/she can use any of the other functionalities. Registration ensures that a proper ID and password are created for each new user.

The first step in handling a given programming assignment for a specific course is for the *Instructor* to *Cre-*

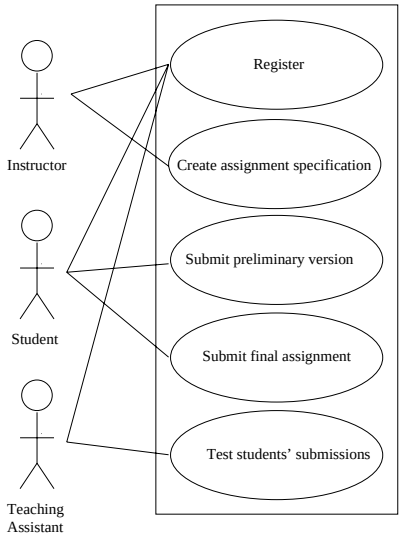

Figure 3. Use case diagrams for core functionalities of OCETJ

ate an assignment specification. [3] The two interesting attributes of such a specification are its public and private test suites, described in more detail below. Other attributes, such as the final submission date, the number of tentative submissions allowed, etc., must also be specified.

In OCETJ, it is the *public* test suite specified by the instructor for a given assignment that is used to provide students with early feedback. More precisely, a *Student* can first use OCETJ to *Submit a preliminary version* of his solution to a programming assignment. The system then tests the student's submission on the public test suite specified by the instructor, and appropriate results are sent back to the student indicating which test cases have succeeded/failed. The detailed content of these public test suites can either be kept private, providing only students with the (meaningfully chosen) names of the failed test cases, or can be made available to the students — this would be especially useful in the early part of an introductory course, to help students identify which part of their code does not work properly. Once the student is satisfied with his solution, he can then *Submit a final assignment*, solution which is then saved in order to be subsequently marked.

Providing early feedback through public test cases should, hopefully, help students improve their solutions and ensure that their submitted solutions *minimally* behave correctly. One possible *danger* is that students rely too much on these tests, without properly defining their own test suite. This problem can be alleviated by having the instructor specify a limit on the maximum number of tentative submissions allowed for any given student. With a more sophisticated tool, the students tests classes themselves could be evaluated, as done by the ASSYST system [13].

Finally, when the final submission date has been reached, the *Teaching Assistant* (which could be the instructor himself) can *Test the students' submissions*. During this latter phase, the system, using the *private* test suite, automatically tests *each* of the submitted solutions, generating reports saved in a database for subsequent use by the instructor or teaching assistant. Of course, the private test suite will generally be much more complete than the public one, testing not only the core behavior of the expected solution, but testing also more complex or limit conditions and behavior.

It is important to stress that the test results do not, by themselves, determine the final mark, but rather are to be used subsequently in conjunction along with other criteria to guide the teaching assistant in the final grading process. Thus, OCETJ attempts to *facilitate* the marking process, not to replace the instructor or teaching assistant.

Implementation of OCETJ

OCTEJ has been designed as a web application and implemented using various Java technology.

The users access the system using web browsers. The web pages and scripts are generated and implemented using Java *servlets* [6] (running on a Linux apache web server). A servlet is a Java program that runs on a web server and plays a role similar to a CGI script: it receives an HTML request from a client web-browser, parses the request, calls some appropriate methods in the application layer, and finally, using the results received from the application, generates a response (HTML) to be displayed by the client browser.

The application logic *per se* is implemented (mostly) using simple transaction scripts [10] written in Java (but see below for a special case). Persistent information is managed using two distinct mechanisms. Special directories accessed through a (Novell) local network are used for the instructors' public and private test suites as well as for the students' assignments. The data created and managed directly by OCTEJ (e.g., registered users, assignment specification attributes, test results) is handled with an Oracle database, accessed using JDBC (Java DataBase Connectivity). The JDBC API allows a Java program to contain embedded SQL operations, making it possible to create tables, add records, make queries, etc.

The testing of the various students' final submission is not done purely within the Java framework. As explained earlier, an executable program is obtained by combining the code submitted by a student together with the test class defined by the instructor, where the latter contains the main program. Testing any given student submission thus requires compiling the code submitted by the student, and then running a Java program composed of the instructor's test class together with the student's compiled code. Furthermore, this process must be repeated, *independently*, for each student. Thus, the compilation and testing of the various students' submission has been implemented using a number of batch (`.bat`) files that somewhat go through

[3] Of course, once created, such an assignment specification can also be modified and, later, deleted. For simplicity, such basic use cases have been omitted from the diagram.

the following process:

1. Copy the student's code to some private directory;
2. Compile the student's code with `javac`;
3. Run the resulting program with `java`;

The batch files' execution is triggered, from within the Java virtual machine running when the servlet is activated, by forking a `Process` using the `exec` method of the `Runtime` class. Note that after each instance is run, the number and name of the tests that failed, if any, are saved in the database; these results can subsequently be used by the instructor or teaching assistant for grading the student's submission.

Even though OCETJ has been designed as a web-like application, its current implementation, for technical and security reasons, only runs on a campus-wide intranet. Thus, the students submit their programs, not by email, but by saving them to a special (protected, *write-once*) directory which can be accessed only by the instructor (or by OCETJ). Future work will involve deploying OCETJ as a pure web application.

5. CONCLUSION

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 OCETJ tool. Two key pedagogical concerns are also addressed by the use of the JUnit 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.

OCTEJ is currently being used for the first time this term, in a second year Java course (technical college degree), although only in a limited way (no submission of preliminary versions by students). The complete implementation should be available to the students next fall.

For future work, we plan to re-implement OCETJ as a pure web application, allowing it to be used in a more general context than its current one. We also plan to examine how various design and source code metrics could be used to provide *quality* metrics, thus providing further objective 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 strong support to help reduce the marking workload.

6. ACKNOWLEDGMENTS

The second author would like to thank his colleagues from the Dépt. d'informatique (Cégep du Vieux-Montréal) for their help and support, as well as Ms. B. Thuot (*technicienne en informatique*) for her precious technical help. Thanks also to L.H. Bouchard et B. Lefebvre (professors, Dépt. d'informatique, UQAM) for their comments on a first draft of this paper.

7. REFERENCES

[1] 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.

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

[3] K. Beck. Aim, fire. *IEEE Software*, 18(6):87–89, 2001.

[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] D.R. Callaway. *Inside servlets: Server-side programming for the Java plaform*. Addison-Wesley, Reading, MA, 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] M. Fowler. *Refactoring — Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.

[10] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, MA, 2003.

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

[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] É. Labonté. Outil de correction semi-automatique de programmes Java. Master's thesis, Dépt. d'Informatique, UQAM, déc. 2002.

[17] C. Larman. *Applying UML and Patterns — An Introduction to Object-Oriented Analysis and Design (Second Edition)*. Prentice-Hall, 2002.

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

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

[20] R.C. Martin. *Agile Software Development — Principles, Patterns, and Practices*. Prentice-Hall, Upper Saddle River, NJ, 2003.

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