Int. J. Engng Ed. Vol. 25, No. 4, pp. 814–829, 2009
Printed in Great Britain.

0949-149X/91 $3.00+0.00
© 2009 TEMPUS Publications.

A Fully Automatic Approach to the Assessment of Programming Assignments*

MARCO TORCHIANO, MAURIZIO MORISIO
Computer and Control Department, Politecnico di Torino, Italy.
E-mail: {marco.torchiano, maurizio.morisio}@polito.it

The (semi-)automated management of programming assignments promises efficiency in large classes and fairness for the students. This paper presents an approach and the relative supporting environment for automated assessment of programming assignments that was developed for an object-oriented programming course. The user interface of the tool is tightly integrated into the Eclipse development environment, which is used by students in the labs. The assessment and grading are based on automated tests that allow an objective evaluation of the working features delivered by the students. The grading is based on the number of test cases passed, and on the distance, measured in lines of code, between the program developed in the lab and a corrected, fully functional version of it. This two-phased approach allows achieving a quantitative evaluation of the defects and of the missing functions.

Keywords: Laboratory-based teaching; automated learning systems; teaching/learning strategies; programming and programming languages; object-oriented programming; Java programming

INTRODUCTION

UNIVERSITY COURSES of object-oriented analysis, design and programming are now mainstream in software engineering and computer science curricula.

The computer science curriculum at the Politecnico di Torino, Italy (three years, roughly corresponding to a B.S. in the American system) includes three programming courses: basic programming (using the C language), algorithms and data structures (using C language with pointers and dynamic memory), object-oriented programming (using Java [1]). These courses each weight five credits in the European ECTS system [2] that is roughly equivalent to 50 hours in the classroom. The authors are involved in the latter course.

All these courses follow a constructivist approach: students attend the lessons, receive assignments that they develop either in the labs, with assistance from senior students, or at home. Eventually they take an exam, which consists in developing a program. Traditionally the program was developed on paper in a traditional classroom (no PCs available) during a session lasting around two hours· After this session the students, using a carbon copy of the program handed to the teacher, developed the program on a PC, completing and debugging it as needed. Next, they had another individual session with the teacher, who graded the student according to several factors: the program developed in the classroom, the program developed on the PC, the differences between them in terms of provided functionality, design choices and defects.

The drawbacks of the paper-based approach have been evident to all teachers for a long time. The basic obstacle has always been the lack of a PC per student in the labs both for logistic and budget limitations. This problem has been solved recently. Therefore, since the 2003 edition of the course, the building of a tool (PoliGrader) was undertaken to automate the delivery and evaluation of assignments. The system has become stable since 2006 and offers the following main features:

- Storage and delivery of assignments
- Validation of assignments (automatic, test based)
- Grading (also automatic, based on the quantitative evaluation of defects)
- Tight integration into the development environment (Eclipse)
- Web-based monitoring and management.

This paper presents our approach to summative (designed to make a judgment about student achievement) and formative (designed to improve student knowledge and skill) automatic assessment of programming assignments. Below we will discuss the tool, called PoliGrader, the corresponding assignment process, and how the educational goals were achieved.

While many features of PoliGrader can be found in other tools, the distinctive feature and our main contribution consists in the quantitative evaluation of the defects and missing functions.

* Accepted 20 March 2009.

CONTEXT AND MOTIVATION

The assignment assessment approach and the relative organization of the laboratories were born out of several factors. First of all, the maturity achieved by the development tools (i.e. Eclipse[3] and Junit [4]) provided the basis to build a stable and robust infrastructure. Thus it became possible targeting a large set of issues deriving from both general considerations and lessons learned in teaching the object-oriented programming course: they are presented below, grouped under major headings.

Local context

During the course, every week the students develop a new assignment, partly in assisted labs and partly on their own. Senior students provide assistance during lab hours, both on use of the tools and on programming and Java topics. The main problem with such an approach lies in the limited number of senior students available, and therefore in the low assistant to student ratio. Ideally, each assignment developed by each student should be evaluated for programming style and functional correctness, but this is not possible with the available resources.

Besides, it appeared that the students considered a program complete as soon as they concluded the coding phase. They did not go through a testing phase, apart from, in some cases, running the program with a sample input and observing the results. Moreover, many students, in the lab session, didn't even go as far as compiling the programs; therefore their programs were full of syntactic errors. A related issue is about students' motivation. Ample anecdotal experience was collected from the paper-based exams; for instance when explaining to the students how to use an Integrated Development Environments (IDE) some students asked: 'Why do we have to learn how to use an IDE when the exam is on paper'?

This negative attitude derives from the habit of handing out programming assignments on paper.

In addition, from the point of view of the students, developing the program on paper is frustrating, especially because they are used to state of the art tool support in the lab sessions, including extensive Java documentation, syntax checks, automatic code completion, pretty printing etc.

Finally; from the point of view of the teacher and the students, grading fairly and consistently, especially when more than one teacher is involved (the course has around 200 students per year), is hard. And the textual description of the program to be developed is often subject to ambiguities or misunderstandings.

Educational goals and constraints

The course builds upon preceding programming courses and focuses on object-oriented programming, while object-oriented analysis and design are taught in a successive software engineering course.

Given the current schedule of the faculty, the course has severe time contraints: it has to fit into a seven-weeks (60 hours) slot, including lectures and labs. The main consequence is that only essential topics find a place in the course and no time is left for design issues: this is inevitably a programming only course.

The educational goals defined for the course are:

- Basic OO concepts and mechanisms. A student who follows successfully the course should master concepts like class, instance, encapsulation, message passing, inheritance and their implementation in Java.
- Exceptions and error handling. A student should master exceptions, as supported by Java, as a mechanism for error handling.
- Collections framework. A student should be able to understand and master the most used data structures and algorithms provided by the Java Collection Framework (such as maps, lists, sorting and searching algorithms).
- Syntactic correctness. A student should be able to write synctactically correct Java programs.
- Functional correctness. A student should learn that a program should be functionally correct, and not only synctactically correct. Functional correctness is not intended as absolute, but as evidenced by success on a number of black box test cases. Apart from being a must in industrial practice, functional correctness has an important pedagogical side effect. In [5] it is suggested that a 'complete solution is regarded as an important step in building the confidence of student programmers, even if some initially complete only the simplest task'.
- Fairness in grading. Of course, all programs of all students should be graded in a fair and repeatable way. Syntactical and functional correctness are the starting point. However, not all defects are equal. A consequence of this is adoption of the two phased grading scheme. The scheme considers, in phase one, the delivered functionality (in terms of number of acceptance tests passed) and, in phase two, the distance from the full functionality requested. As a proxy of this distance the number of lines added or changed from phase one to phase two is used. The grading process is described in detail below.

The capability to write effective test cases [6] [7] could be another educational goal linked to this course, but the limited time factor has excluded it. Object-oriented style is another key goal. A student should be able to write programs in object-oriented style, i.e. with minimum coupling between classes, with maximum encapsulation and correct use of delegation. To state it on the negative side, a program written by a student should not have 'bad smells', as defined in [8]. Automatic check of bad smells via Eclipse plug-ins is currently being added to PoliGrader.

System requirements

According to [9] there are three main components in the assessment and grading process: correctness, style and authenticity. In addition, the authors identify a fourth component that is essential to achieve fairness: error severity. The reasons why a requirement in the assignment is not satisfied by a student's program may be very different; it is necessary to evaluate objectively the error (or omission) committed by the student in order to assign fair grades.

In the context in which the PoliGrader approach was originated, style is difficult to evaluate due to the small size of the programs; authenticity is a minor issue because, during the exams, a number of teaching assistants patrol the lab to reduce plagiarism and communication. Thus the main focus is on correctness and error severity.

The issues and constraints discussed so far can be summarized in the following requirements for an assessment approach:

- provide automated support for the lab assignments (delivery to students and assessment)
- provide automated support for the final exam (delivery to students, assessment and grading)
- support an automated grading process that allows evaluation of error severity
- support monitoring of all steps in the process: definition and scheduling of assignments, statistics on assignments evaluated, passed by students, etc.
- integrate all functions available to the student in the IDE (Eclipse in this case).

RELATED WORK

The history of computer-based approaches to handling, assessing and grading assignments for students is now at least 20 years old. We consider the related work from the two perspectives that are of more concern from our point of view:

1) the tools and techniques used to handle and manipulate the assignments
2) criteria adopted for grading the assignments.

Tools

Table 1 reports a list of the main tools aimed at supporting exams and laboratories in computer science education. The table presents the tools (columns) and the features provided (rows). The main features identified are storage and delivery of assignments to students, assessment of assignments, results of assessment, grading. An X in a cell indicates that the tool in the column supports the feature in the row. The bottom row reports the programming language supported.

Ceilidh [10] is one of the first tools described in the literature; it is a general purpose tool capable of handling different kinds of assignments and courses, and different roles (teacher, student, etc.). Grading is achieved via different marking tools.

CourseMarker [11] is an evolution of Ceilidh, mostly in terms of the support system software architecture. Further, it adds more flexibility in customizing and grading of assignments.

ASAP [12] has similar functions, originally started for C++ programs; later it was provided with Java support.

BOSS [8] was the first system to integrate JUnit for testing Java assignments.

RoboProf [13] is a web-based application for administering assignments to students and grading them in function of the number of test cases passed. It has been used to teach Java to first year students, both with small assignments during the course, and with longer assignments in the final test. The author demonstrates that

Table 1. Computer-based systems for student assignments

	PoliGrader	**CourseMarker**	**Ceilidh**	**ASAP**	**BOSS**	**RoboProf**	**WEB-CAT**	**JEWL**	**ASSYST**	**TRAKLA2**	**PILOT**
visual algorithm simulation										×	×
storage, delivery of assignments	×	×		×	×	×		×			
validation/test of assignments	×	×	×	×	×	×	×	×	×	×	
test execution results	×	×		×	×	×	×				
grading	×	×	×	×	×	×	×		×		
programming language	java	java, others	java	C++ Java after 2004	C, Java from 2003 (v 2.3)	java	java, others	GUIs	C	NA	NA

Table 2. Grading capabilities

Criteria\Tools	PoliGrader	AutoGrader	CourseMarker	Ceilidh	ASAP	BOSS	JEWL	ASSYST
Fully automated	×	×	×	×	–	–	–	–
Functional correctness	×	×	×	×	×	×	×	×
Defect weight/error severity	×	–	–	–	–	–	–	–
Code quality	–	×	×	–	–	×	–	×
Non-functional properties	–	–	×	×	–	–	–	×

students performing best during the course, also obtain higher grades in the final exam.

WEB-CAT [14] has the usual functions for managing and grading assignments. As well, it takes into account tests written by the students, in particular the WEB-CAT Grader plug-in grades assignments that consider how well students have tested their program (instead of using a black box approach for testing, as made by all other approaches listed here).

JEWL [15] concentrates on definition and testing of GUIs.

ASSYST [16], aimed at C, tries to provide meaningful reports to students about the assessment.

Trakla, its evolution Trakla2 [17], and Pilot [18] support the definition and analysis of algorithms by visualizing the underlying data structures and their evolution.

The tool, PoliGrader (in the leftmost column) was started in 2003; at the time no tool provided satisfactory support for Java programs. PoliGrader supports two roles, student and teacher, multiple courses and specializes in Java assignments. The teacher defines an assignment for a course and a period of time. As students log in they get the current assignment for their course automatically downloaded to their PCs, then they work on it, and eventually submit it. The whole procedure is carried on within the IDE (Eclipse). When the teacher triggers the assessment, the system sends the results to the students, via e-mail.

The distinguishing features of PoliGrader are:

- Integration in the IDE used by the students. Upload and download of assignments are implemented via Eclipse plug-ins
- JUnit tests are a first class entity in the process
- Grading based on the evaluation of the severity of errors.

It is worth noting a number of approaches and tools that assign and evaluate questions (yes, no, multiple closed answers, open answers) or problems (expressed as a text or picture, with related answer). Such approaches and tools can be seen as more general than the ones we have considered above, where the problem assigned requires writing a computer program, which cannot be simply compared to a reference 'right' answer. Apart from the commercial offerings, there are many academic proposals, such as OASIS [19] used in electronic courses, [20] mechanics courses and [21] physics courses. AulaWeb [22] uses a similar approach, but questions are relative to computer science topics, including TurboPascal programs. AulaWeb can be considered the ideal link between the two categories of approaches, questions vs. programming problems.

Grading

An important issue for both teachers and student, and one of the most frequently asked questions in the classroom, is how exams are graded. Different tools adopt different approaches to the grading, as summarized as in Table 2.

Practically all the tools include functional correctness as the main grading criterion. Roughly half of them include some measure of code quality. As far as the authors know PoliGrader is the only one that focuses on measuring the distance between an initial version developed by the student

Table 3. Comparison

Criteria\Tools	PoliGrader	AutoGrader	CourseMarker	Ceilidh	ASAP	BOSS	JEWL	ASSYST
automated support lab assignment	×	×	×	×	×	×	×	×
automated support final exam	×	×	×	×	×	×	×	×
automated grading process	×	×	×	×				
evaluation of defect weight/error severity	×							
IDE (Eclipse) integration	×							

(V1) and a correct, working version (V2). Instead, many other approaches do not provide the students with test cases but only with the test outcomes and allow multiple submissions.

The distinguishing features of PoliGrader are:

- JUnit test cases delivered to the students
- Students use tests to correct and complete their program.

Comparison

We recap in Table 3 the main functions and properties of PoliGrader, in comparison with similar tools. The main differences lay in the two-phased process for correction of assignments, devised in order to evaluate error severity. And the integration in Eclipse IDE, to allow students complete all operations from the same tool.

However, the main reason why we started development of a new tool, instead of using or adapting an existing one, is that in 2003, when development started, other tools were not yet published.

THE CORE PROCESS

The core process presented in Fig. 1 provides a high level view of the approach. Variants of this process, supporting tools and techniques will be explained later. The process has four basic steps:

1) The teacher defines the program to be developed, the assignment definition consists of three parts:
   - a textual description of the program functions and constraints (see Fig. 4 for an example).
   - skeleton code (under the form of Java classes

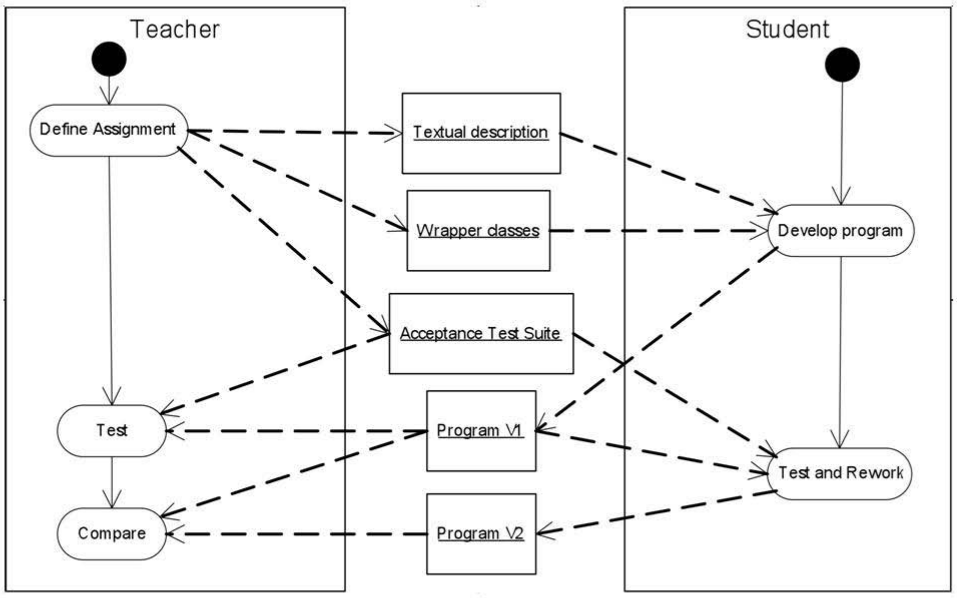


Fig. 1. The core process.

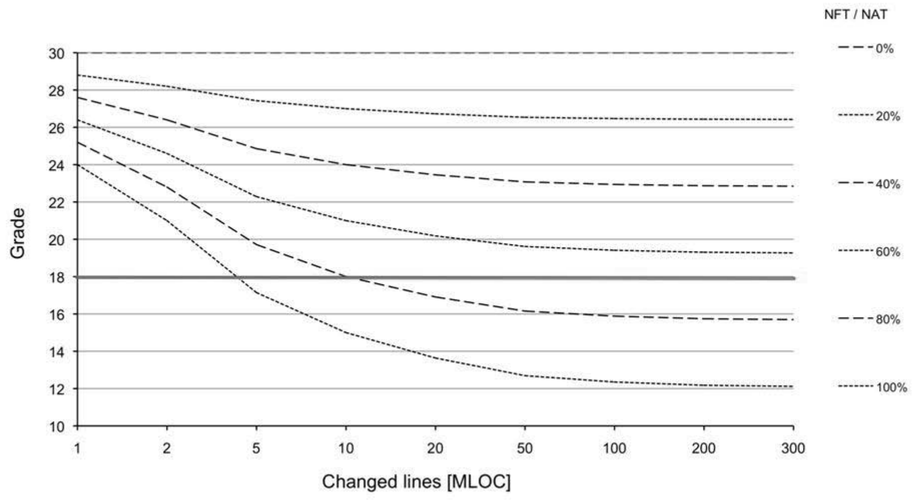


Fig. 2. The grading function.

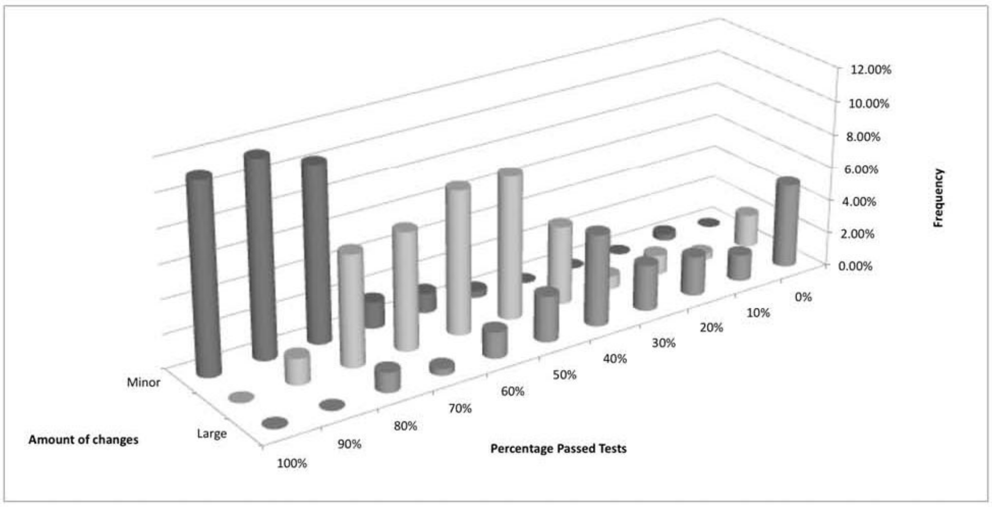
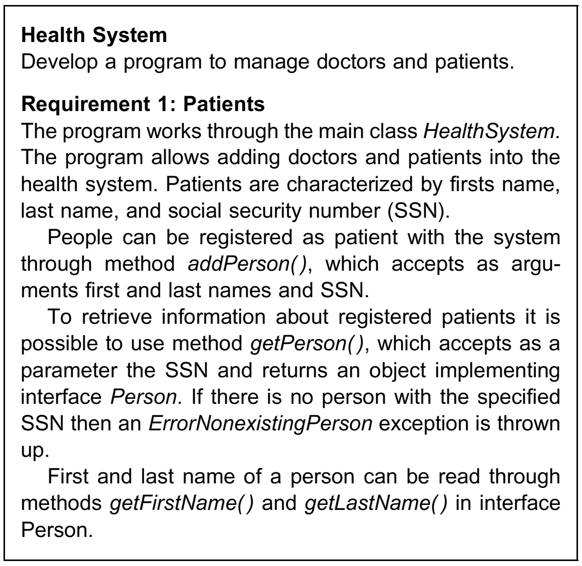


Fig. 3. Frequency of Passed Tests and Changed Lines combinations.



Health System
Develop a program to manage doctors and patients.

Requirement 1: Patients
The program works through the main class *HealthSystem*. The program allows adding doctors and patients into the health system. Patients are characterized by firsts name, last name, and social security number (SSN).

People can be registered as patient with the system through method *addPerson()*, which accepts as arguments first and last names and SSN.

To retrieve information about registered patients it is possible to use method *getPerson()*, which accepts as a parameter the SSN and returns an object implementing interface *Person*. If there is no person with the specified SSN then an *ErrorNonexistingPerson* exception is thrown up.

First and last name of a person can be read through methods *getFirstName()* and *getLastName()* in interface Person.

Fig. 4. Example of textual description.

with prototypes of public methods, or as Java interfaces) (see Fig. 3).

- a suite of black box acceptance tests for the program (under the form of one or more test classes in JUnit format) (see Fig. 4).

Obviously the three parts of the program must be consistent. Acceptance tests call public methods of the wrapper class, and the textual description of the program must match the public functions of the wrapper class. In case of conflicts the part written in Java (wrapper class and acceptance tests) overrides the textual description.

2) The student develops the program in the lab and delivers an initial version (V1) starting from the textual description and the skeleton code.
3) After the end of the exam lab session, the test suite is made available to the student together with the results of the tests applied to his/her V1 program.
4) The student is asked to modify this program until all tests pass. Tests can be run locally as a JUnit test suite. The correct program is delivered as version V2.
5) The student is evaluated taking into account both V1 and V2.

During the course, the students perform lab assignments following the same process except for step 4, which is not mandatory. During the exam all steps are required and the program is graded in function of tests passed by V1 and modifications made to get a full working V2.

Assignment assessment and grading

The assessment and grading of the assignment is made in a quantitative way, based on a set of defined metrics [23]. The metrics are based on the acceptance test suite developed by the teachers and are summarized in Table 4.

Table 4: Assessment metrics

Metric	Definition
NAT	Number of test methods defined in the Acceptance Test Suite
NPT_s	Number of acceptance tests passed by the program V1 developed by student *s*
NFT_s	Number of acceptance tests failed by the program V1 developed by student *s*. Derived measure = $NAT–NPT_s$
$PASSED_s$	$\frac{NPT_s}{NAT}$ Proportion of tests passed by program V1
MLOC*s*	Modified lines of code in Program V2 with respect to Program V1 developed by student *s*

The functionality of a program (PASSED) is evaluated in terms of passed acceptance tests. Because the teacher develops the Acceptance Test Suite independently, it represents a proxy of the field operation of the program.

The number of modified lines of code (MLOC) represents a proxy of the rework effort and of the severity of errors. The use of a proxy, though subject to errors, is the only possible objective measure.

The grading formula used in the proposed approach is:

$$Grade = Offset + (Max - Offset)\cdot$$
$$\left(1 - \frac{NFT}{NAT}\cdot\left(1 - \frac{Bonus}{(MLOC + Bonus)}\right)\right)$$

The grading formula has two components:

1) a linear component that decreases with the number of failed tests
2) a hyperbolic component that decreases with the number of changes (in terms of lines of code changed).

In the grading formula two parameters are used for fine tuning:

1) the offset
2) the bonus.

The offset is a scale factor, which sets the maximum number of failures allowed to achieve a sufficient grade (18 in the Italian grading system) even in the worst case (i.e. very severe errors).

The bonus is a slope factor that defines the relative importance of delivering fully working single features with respect to achieving an almost overall complete solution (with possibly fewer requirements fully implemented).

The implication of this grading schema on the grades achieved by the students can be better understood by looking at the graph presented in Figure 5. In the Italian University system grades are in the range [0–30], an exam is failed if the grade is < 18. An A in the US system corresponds roughly to a grade in the range 27–30. The figure presents the grade achieved in function of the number of lines changed between the two versions of the program (MLOC). Each curve corresponds to a different number of tests failed by the first version of the program (NFT).

For instance it can be observed that for any number of failed tests, the less the lines changed, the higher the grade. In general for small number of tests failed, the influence of the number of changed lines is limited.

In theory, a student who passed all the tests but one (i.e. one failed test) even with 300 lines of code changed will achieve at least a grade of 28.5. In practice, if just one test fails the changes required to fix this problem will be likely at most of 5–10 lines. In other terms, not all the domain of the grading function (see Fig. 6) is equally probable.

The domain of the grading function is represented by the product of NFT and MLOC. The actual frequency of the points in the domain is plotted in Figure 6, based on 260 students from three different courses. Students who made minor changes are mostly those who passed most of the tests. Students who made many changes are those who failed more than half the tests. Obviously we also observe exceptions: a few students whose program passed few tests, who managed to fix their program with minor changes, a small but high impact defect. However, some students whose program passed a most of the tests had to implement large changes: small impact but large defect.

The extreme case is considered here of a student that developed an almost complete solution with one single error (on a single line, e.g. the missing initialization of a variable), that causes all tests to fail. With a single line modification the program

```
package health;
import java.io.IOException;

public class HealthSystem {
  public void addPerson(String first, String last, String ssn) {
  }
  public Person getPerson(String ssn) throws ErrorNonexistingPerson {
  return null;
  }
  public void addDoctor(String id, String first, String last, String ssn){
  }
  public Doctor getDoctor(String id) throws ErrorNonexistingDoctor {
  return null;
  }
  public void assignDoctor(String id, String ssn)
  throws ErrorNonexistingPerson , ErrorNonexistingDoctor {
  }
  public int loadData(String fileName) throws IOException{
  return 0;
  }
}
```

Fig. 5. Example of wrapper class.

```
public class AcceptHealth extends TestCase {
  public void testPerson(){
    HealthSystem hs=new HealthSystem();
    hs.addPerson(
          "John", // name
          "Smith",// surname
          "123-45-6789"); // SSN
    try{
       Person p = hs.getPerson("123-45-6789");
       assertEquals("John",p.getFirstName());
       assertEquals("Smith", p. getLastName());
    }catch(ErrorNonexistingPerson e){
          fail("Smith should exist.");
    }

try{
          Person p2 = hs.getPerson("111-22-3333");
          fail("SSN should not exist ");
}catch(ErrorNonexistingPerson e){
          assertTrue(true); /* OK */
}
```

Fig. 6. Excerpt from an acceptance test suite.

would pass all tests. The question is how should such a student be graded? It is important to observe how this decision can be expressed by means of the two grading parameters. The grades achieved by such a student in four different grading scenarios (combination of Bonus and Offset parameters) are presented in Table 5.

If more weight is given to the presence of working solutions, the case in the exam will be graded very low (20 – 22.5). On the contrary, if working solutions are not so important and the only aspect considered is how close to the final complete solution the student gets, then the result is a higher grade (26 – 27).

An example

To better understand how the proposed approach works here is an example of an assignment, as experienced by a student. Figure 4 shows a textual description of a program, in this case a trivial version of a health information system.

Along with the textual description, the student receives a wrapper class, (see Fig. **3** above). *HealthSystem* is the wrapper class with all requested functions defined, such as *getPerson()*, and exceptions, such as *ErrorNonexistingPerson*. The wrapper pattern is used to allow the students using the internal design they prefer. Having received the wrapper class and the textual description, the student develops V1 of his/her program.

Then the student receives the acceptance test suite in JUnit format, (see Fig. 6. *AcceptHealth* is the test class that contains test methods, such as *testPerson()*. *testPerson()* tests if the program is capable of adding a person, finding a person that has been defined or raising an exception if the person has not been defined. The student has to improve his/her program (version V2) until it passes all tests.

SUPPORTING TOOL SUITE

A tool suite that supports the enactment of the process described above is available. The tool suite is based on the Eclipse platform. Eclipse is used throughout all object-oriented programming courses at the Politecnico di Torino it was extended it to provide some simple features to support the automated assessment approach.

The supporting environment requires a server-side application to perform authentication and store the assignments, it was developed as a web application upon the Apache Tomcat platform.

Figure 7 shows a summary of how the environment supports the process.

The student sitting at the lab workstation runs Eclipse. A plug-in that provides the authentication features was developed.

When the student authenticates, the information is sent back to the server that replies with both an authorization and the assignment.

Table 5. Extreme cases in grading function

		Importance of working solution over completeness	
		High (Bonus=1)	**Low (Bonus = 4)**
Grade Variation	High (Offset=10)	20	26
	Low (Offset=15)	22.5	27

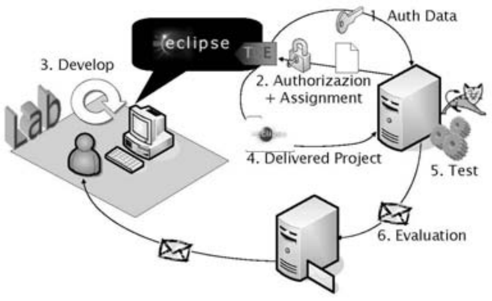


Fig. 7. Support system overview.

When the student has completed the assignment (or the time for the exam has run out) he/she can click a button on the (extended) Eclipse GUI to submit it for evaluation.

The evaluation is usually performed offline and the results of the tests, together with the acceptance tests themselves, are sent to the student via email.

The system has two categories of users: teachers and students. The functions provided to the system users can be accessed through two distinct access modalities: either within the Eclipse IDE or through the web by means of a web browser. The Eclipse modality requires the installation of a small plug-in (already installed in all labs at Politecnico di Torino). Table 6 shows an overview of the functions provided by the system, the user using them and the modality of access.

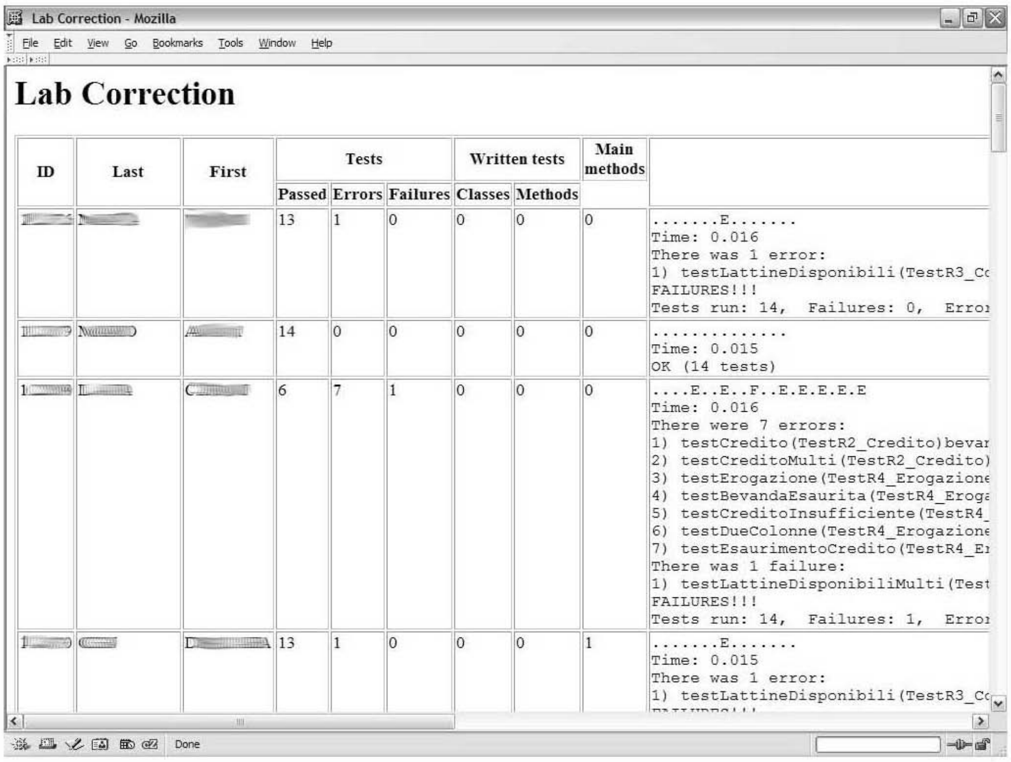
Students interact with the system though the Eclipse IDE while in the lab, while they access it using web browser when at home.

Teachers access the system solely through a web-based interface. As an example Fig. 8 shows the results of a lab session assessment: tests are run on the projects submitted from the lab by the students, a full report is generated with the results from the tests.

Particularly relevant is the exam grading function, where the system takes projects submitted in the lab and the corresponding projects submitted

Table 6. Main functions provided by the system

Modality	Teacher	Student
Eclipse		Lab login and assignment download
		Lab assignment delivery
Web	Student data management	Home login
	Courses, lab and exam session management	Home assignment upload
	Assignment schedule definition	
	Lab or Exam assessment and grading	



Lab Correction

ID	Last	First	Tests			Written tests		Main methods	
			Passed	Errors	Failures	Classes	Methods		
[redacted]	[redacted]	[redacted]	13	1	0	0	0	0	`.......E.......` `Time: 0.016` `There was 1 error:` `1) testLattineDisponibili(TestR3_C` `FAILURES!!!` `Tests run: 14,  Failures: 0,  Erro`
[redacted]	[redacted]	[redacted]	14	0	0	0	0	0	`..............` `Time: 0.015` `OK (14 tests)`
[redacted]	[redacted]	[redacted]	6	7	1	0	0	0	`....E..E..F..E.E.E.E.E` `Time: 0.016` `There were 7 errors:` `1) testCredito(TestR2_Credito)beva` `2) testCreditoMulti(TestR2_Credito)` `3) testErogazione(TestR4_Erogazion` `4) testBevandaEsaurita(TestR4_Erog` `5) testCreditoInsufficiente(TestR4_` `6) testDueColonne(TestR4_Erogazion` `7) testEsaurimentoCredito(TestR4_E` `There was 1 failure:` `1) testLattineDisponibiliMulti(Tes` `FAILURES!!!` `Tests run: 14,  Failures: 1,  Erro`
[redacted]	[redacted]	[redacted]	13	1	0	0	0	1	`.......E.......` `Time: 0.015` `There was 1 error:` `1) testLattineDisponibili(TestR3_C`

Fig. 8. Teacher web interface (ID and names of students have been erased).

from home and applies the grading algorithm described earlier. The result consists in a spreadsheet with all the data about the projects and three free parameters: Offset, Bonus and Max. It is possible to operate on these parameters to adjust the grade spectrum.

For each student who submitted a solution for the assignment, we have the following information:

- Student reference: ID, First and Last name
- Metrics of acceptance tests results:
- NPT_s or number of passed acceptance tests,
  a) Number of tests terminated by errors (an error being an unexpected,
  b) nullpoint exception),
  c) NAT—NPT_s or number of failed acceptance tests.
- Product test statistics for the submitted program:
  a) Number of test classes and test methods written by the student, indicating that he/she tested the program,
  b) Number of main methods written by the student, indicating that he/she ran the program.
- A summary of the acceptance tests, a short output from JUnit documenting which tests failed or caused an error.

Projects submitted by students in the form of compressed files present several problems when it comes to automatic processing. The most common issues are:

- the root of the project may not match the root of the compressed file (sub-)system
- the project may contain only part of the bytecode or even only source files
- the project may already contain the tests we want to run possibly modified (or hacked) by the students
- the student may have changed some method signature
- the project may contain deadlocks or endless loops
- the project may terminate the execution with a System.exit() statement, that terminates the virtual machine and therefore the whole evaluation process.

To address these possible problems an automatic testing procedure was designed together with the relative testing harness based on JUnit.

The procedure consists in the following three steps:

1) find Java project root (.project, .class files)
2) (re)compile source files
3) run tests.

The first step is to apply some heuristics in order to identify the root of the Java project, which may not coincide with the root of the file system portion contained in the compressed file. Initially the 'project' file that is created by the Eclipse environment in the root of the project is sought; however, it happens that if the project is wrongly imported or exported (to/from Eclipse) many such files may end in the compressed file system. If it is not possible to find a root in such a way, then source files are parsed to extract the package declaration and deduce the root location as a relative path with reference to the source file position. In this case a majority voting is used if several sources point to different roots.

The second step consists in (re-)compiling the source code.

Finally, tests can be run. An ad-hoc environment is required to address the issues listed above. In particular the system takes the following 'counter-measures':

- tests are located in a specific hard-to-guess package
- tests have been compiled in advance, once and separately from the students code
- a watchdog is used to avoid endless loops or deadlocks
- a security manager avoids undesired or malicious behavior.

The use of a specific package practically avoids the risk of using student-hacked test cases.

Since the test code is pre-compiled, the cases of students who changed the signature of methods invoked by tests can be handled gracefully: when a method required by a test case is missing, a NoSuchMethodException exception and the resulting test failure are observed. In addition since tests are not recompiled for every student, a significant performance enhancement can be achieved.

The presence of a watchdog in a parallel thread allows forcing an exception in the testing thread, therefore deadlocks and endless loops are reported as errors to the user.

It is possible that student code contains calls to 'dangerous' methods, e.g. System.exit() that would terminate the VM or accessing system files on the testing server. During test execution a security manager is activated that prohibits such behaviour and generates an exception instead.

ASSESSMENT OF THE APPROACH

Assessment of the proposed approach can be conducted from two points of view: teacher and student. From the teacher's point of view, it is important to verify how the approach helps to attain stated educational goals. From the student's point of view the focus is on perception of the new approach with respect to the more traditional paper-based approach.

The teacher's point of view: educational goals

Table 7 summarizes the educational goals of the course, the measures proposed to evaluate their achievement, and their results for academic year 2007/2008. The results are relative to programs

Table 7. Educational goals and their achievement

Teaching goal	Teaching sub-goal	Measure of achievement	Result
Basic OO concepts	class, instance, constructor, message passing	classes defined and used by student	98% of students define and use own classes
	visibility rules	number of public data members	95% of students do not define public data members
	inheritance, interfaces	see sorting and exceptions	
Exceptions	exceptions, try, throw, catch	test cases on exceptions	80% of students pass exception test
Collections	List, Map	number of List Map used	82% of students use at least a Map or List
	Search, sort, comparable, comparator	test cases on sorting	45% of students pass at least one test on sorting
Syntactic correctness		no compilation errors	95% of student deliver a program without syntactic errors
Functional correctness		test cases passed	4% of students deliver a 100% working program, 60% deliver a 50% working program
Fairness in grading		LOCs changed to correct the program	25% of students don't submit the corrected program.
			Around 5% of students are not satisfied with the grade.
			The others on average change 52 LOCs.

developed by students in the first two exam sessions following a course.

The first teaching objective relates to basics of object orientation. For concepts such as class, instance, constructor, message passing we evaluate if a student has learned these concepts by checking if he or she has defined and used at least one class. This implies knowing what a class is, defining and using a constructor, sending messages to instances of the class. In practice all students achieve this goal (98%).

A related teaching goal is to know and apply visibility rules. The teachers insist on the need to encapsulate data members of classes. As a measure of achievement we verify if students define public data members. This happens for 5% of students. Overall the goal is achieved. Nonetheless, the fact that some students define public members (and probably use a global data programming style) is discouraging, since the teachers explicitly remark that data members should never be public.

Students should learn to define and use exceptions. Most functions requested of students are tested via three to four test cases. These test cases are mostly about nominal cases, while one or two are about error cases handled via exceptions. If a student passes a test case about an error condition, he/she is able to use exceptions. 80% of students pass test cases relative to exceptions.

Students are encouraged to use collections to implement the data structures needed. All programs requested require data structures that can be implemented in terms of lists and maps. Students have an inherent advantage using collections, because the time available to develop the program during the exam is not sufficient to develop their own data structures. So students who develop all the functionality requested usually master the use of collections. As a measure of achievement we count the number of lists and maps used in a program. 82% of students use at least one of them. Typically, students who do not use them use the more familiar arrays. They are also the ones who score less on number of test cases passed.

All exam programs require some kind of sorting. Specific test cases are provided to test the sorting functionality. 45% of students pass at least one of these tests. This lower figure may be due to the greater difficulty of learning sorting and interfaces (Comparable, Comparator). But also to the fact that sorting functions are typically developed at the end of the assignment, when all basic functions are in place. So shortage of time may also be a reasonable explanation.

Inheritance is another key educational goal. Sometimes a program requires the definition of subclasses (e.g. class Vehicle, class Car, class MotorBike etc). However, given the short amount of time available and the small size of programs developed, this case is not frequent. Nevertheless, students have to use inheritance indirectly, while using Java services, like collections and exceptions. For these reasons achievements on exceptions and collections are considered as proxies of achievements on inheritance.

Syntactic correctness is easily evaluated on the number of syntactic errors in a program. Only few students (5%) fail to achieve this goal. Sometimes this is due to lack of time to complete the assignment, and lack of planning on how to complete it. But usually the reason is lousy preparation of the exam.

Functional correctness is a key goal of the course. It is tested by test cases that are written by the teachers to verify functional coverage. Each function required is covered by 2–4 test cases, and the association test cases—function is underlined to the students. So they learn that syntactic correctness is not enough, a function is complete and correct if it works in a number of situations as formalized by the test cases. A few students achieve full coverage (4% of students pass all test cases), but 60% pass at least half of the test cases.

Finally, fairness in grading is considered. The grading mechanism currently in place is objective and repeatable. It is also open, in the sense that the information (tests passed and not passed, LOC changed, grades) can be accessed, compared and checked by the students. From these points of view the approach is clearly superior to the paper-based approach, especially because in that case the only public information available to students is the final grade. The open issue is whether the measure used is unfair in some specific cases. In this regard around 5% of students complain or want to discuss the grade obtained. This figure is in line with what the authors and their colleagues experience in other exams. Out of this 5% of cases further discussed and analyzed, the grade is changed in a few cases. The most recurring case is a student who fails to pass some tests (say five) and who has to change several lines (say 30). Out of these lines, one line fixes four tests. In this case the grading algorithm penalizes the student, because all lines carry the same weight.

Overall teaching objectives are achieved to a greater or lesser degree. But in all cases the PoliGrader tool and approach is instrumental in evaluating how and to what extent these objectives are achieved. Notably the two-phase grading considerably improves the fairness, as compared with the previous approach.

Student's point of view: course acceptance

Two different evaluations have been conducted: one comparing the same course before and after the introduction of the new approach, and another comparing two similar programming courses using the proposed approach and a traditional paper-based approach.

First, the courses held in 2003 are compared with those held in 2004. In the former, labs and exams were managed in the traditional way; in the latter labs and exams were managed for the first time with the approach presented in this paper.

Students were requested to fill in an evaluation form at the end of each course. The form is anonymous, has the same questions for all courses and is processed by the university teaching evaluation unit. Out of the 10 questions in the form, one addresses the labs.

The average score on this question was 2.6 (on a scale 1 to 4) in 2003. It increased to 3.4 in 2004; this difference is statistically significant ($p < 0.001$). All other factors (labs, PCs, number of teaching assistants) not having changed, we assume this variation is due to the availability of the new process to develop and test assignments in the labs. Direct discussion with students confirmed that they were quite happy with the development environment and the way they could evaluate their programs.

As an additional evaluation it is possible to compare two closely related programming courses: the object oriented programming (OOP) course (where the new approach is applied) and the algorithms and advanced programming (APA) course (where the previous paper-based approach is still in use).

A questionnaire was given to the students to assess their perception of the new approach compared to the old one. The detailed questions are presented in Table 8.

The answers to questions Q1 through Q4 are summarized in Fig. 9. The difference in terms of perceived usefulness is tested. The perceived usefulness of OOP lab procedure is significantly

Table 8. Comparison questionnaire

ID	Question	Scale
Q1	The APA lab approach (program written on paper) supports learning well.	5 points Likert scale
Q2	The APA exam approach (program written on paper) supports learning well.	
Q3	The OOP lab approach (program developed on PC) supports learning well.	
Q4	The OOP exam approach (program developed on PC) supports learning well.	
Q5	Which approach is most suitable to support learning programming techniques and basic concepts?	pc-based development with automatic correction;—don't know;—paper-based approach
Q6	Which approach is most suitable to achieve good results in exams?	

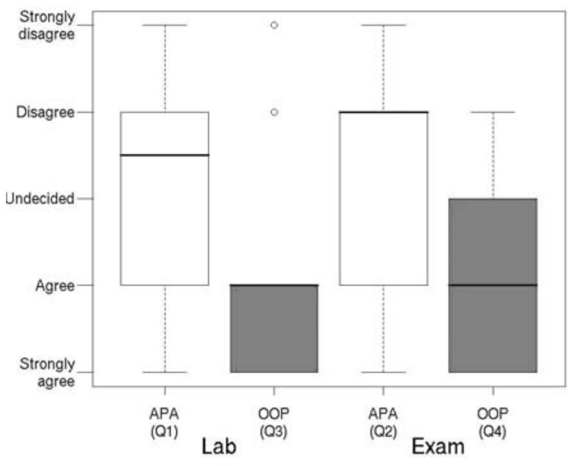


Fig. 9. Answers to questions Q1–Q4.

higher than APA ($p < 0.001$) by one point in the 5-point scale. The usefulness of OOP exam procedure is significantly higher than APA procedure ($p < 0.001$) by two points in the 5-point scale. The difference remains significant also when opinions about lab and exams are merged together ($p < 0.001$).

Questions Q5 and Q6 propose the same concept but in general and not on a specific exam. Answers are summarized in Fig. 10. Also in this case the PC based approach resulted largely preferred both for labs ($p < 0.001$) and exams ($p < 0.001$). Particularly, an average 82% of students found that the PC based approach in the labs supports learning better that the traditional paper based approach; while 77 % thought that the PC-based approach in the exams can lead to better results.

Open issues and future work

Continuous use of the PoliGrader approach over several academic years allowed us to identify the following problems and issues:

- Evaluation of the quality of elaborates of students.

  The acceptance tests developed by the teachers are, by definition, black box and must not make any assumption on the internals of what students produce. The white box evaluation of student's assignment (high level design issues such as choice of classes and functions, correct use of inheritance, use or misuse of patterns, simplicity; low level design issues such as choice of algorithms and data structures; formal issues such as names of classes and functions, indentation and documentation), if required, has to be done manually. It is possible to automate, at least partially, white box evaluation. For instance UML class diagrams can be obtained automatically and design metrics can be used to evaluate design. However, a problem here is the low maturity of tools and techniques available, although a number of Eclipse plug-ins are emerging that can provide immediate feedback to the programmer about the quality of her code.
- Misunderstandings in the program description

  The use of the wrapper class has reduced student misunderstanding of the functionality they are requested to develop. However, text in natural language is still used to describe the functions and in some cases misunderstandings continue to happen. The provision of a full set of acceptance test cases may be a solution; it is planned for the future.
- Design freedom, wrapper class.

  Teachers define a wrapper class (or sometimes a Java interface) in order to define a single set of acceptance tests for all programs developed by all students. Students must develop their program within the wrapper class and never modify it. In case of even the slightest modification compilation will fail and no test will pass. This may be considered a constraint on the design of the program developed by the students; in practice this is a minor issue compared to the advantages of the approach. Besides, similar constraints are typical in commercial development. And finally students have full freedom of choice within the wrapper class.
- Effort to develop acceptance tests.

  Using this approach requires an upfront investment in developing the tool infrastructure. Besides, at every new exam session, the program assigned to students must be developed and the corresponding acceptance test cases defined. Writing the test cases usually requires several hours. Assuming five hours to write the test cases (for programs featuring 4–6 classes, to be developed in 2–4 hours) and assuming that an exam in the new form takes 20 minutes vs. 40 in the old form, break even lies at 15 students.
- Quantitative assessment of course, students and programs

  A side-effect of this approach is the availability of a growing dataset of programs, from assignments in the labs and in exams, from a large population of students. This allows studying quantitatively issues such as quality of programs, productivity of students.

  It also allows a baseline of measures be devel-

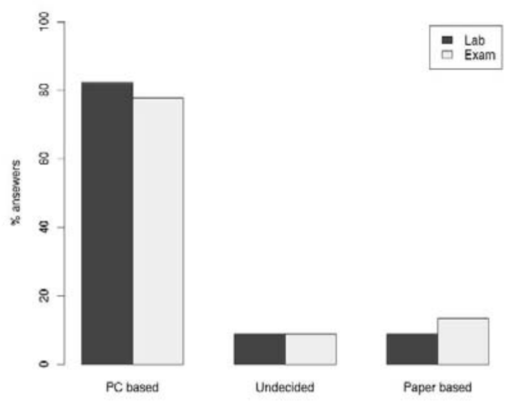


Fig. 10. Answers to questions Q5–Q6.

oped to characterize the student population, to compare it with professional developers, and to use it for experiments with student subjects. The measure baseline can thus be used to assess the course itself: effectiveness of teaching techniques and teaching materials, performance of teachers and students.

And finally, the assignments themselves can be measured. Ideally, all assignments (especially for the final exams) should be of similar size and complexity. Two quantitative means are now available to assess them. A priori there are the number of acceptance test cases, which can be considered a proxy of size and complexity of a program. However, this measure may be biased because acceptance tests are defined by the teacher. A-posteriori there is the difference in size between the programs V2 and V1. If this difference is larger than average, the program may be more complex or longer than average. In these cases the grading can be adjusted accordingly.

CONCLUSIONS

This paper presented an approach for the automated assessment of programming assignments, which is made up of a process and a tool-suite. An assignment has the form of a textual description of a programming problem, and a suite of acceptance tests. The tools support the download of assignments from students, the upload of programs to a testing server, the testing and grading of programs developed by students, the monitoring of all these activities by the teachers.

The evaluation process consists in three main steps:

1) A student develops the required program, uploads it to a server that executes the test suite,
2) Based on the failed tests (if any) the student completes or improves his program untilall tests pass,
3) Using the number of tests failed, and the distance between the initial and the final program, a grade is computed.

From an educational point of view, this approach stresses the importance for the student of developing a working program. The student is exposed to a variety of acceptance test cases that stimulate his program in ways that may not have been considered. Besides, the approach stresses the importance of incremental improvements until a program passes all tests. The measure of the number of changes performed is used to grade the program, thus balancing the sharpness of the outcome of each test case that either passes or fails.

In summary, the specific and novel features of this approach are the following:

- grading is based on two versions of the program, allowing an objective and quantitative evaluation of defects and missing functions,
- the student's user interface is tightly integrated in the Eclipse development environment, thus there is no disruption between the development phase and the assignment submission,
- the grading scheme allows fine-tuned balancing between importance of fully working solutions versus almost complete programs.

Finally, the students appreciate this new approach. Their satisfaction, as measured by questionnaires, is higher when the labs are organized around the automatic assessment approach than the previous paper-based version.

Acknowledgments—We wish to thank Giorgio Bruno, teacher at Politecnico di Torino for his cooperation and Hakan Erdogmus for the useful discussions that led to the definition of the measures and tool infrastructure of the course.

REFERENCES

1. K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*, Addison-Wesley (2000).
2. European Commission, *ECTS—European Credit Transfer and Accumulation System.* Available at: http://ec.europa.eu/education/programmes/socrates/ects/index_en.html (last visited on June, 2007). (2006).
3. Eclipse Consortium, *Eclipse Platform Technical Overview*, Object Technology International (2003).
4. V. Massol and T. Husted, *JUnit in Action*, Manning Publications (2003).
5. C. Douce, D. Livingstone and J. Orwell, Automatic Test-Based Assessment of Programming: A Review. *ACM J. Educ. Res. Comp.*, **5**(3), 2005.
6. S. H. Edwards, *Rethinking Computer Science Education from a Test-first Perspective*. In proc. OOPSLA '03, (2003), pp. 148–155.
7. S. H. Edwards, *Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action.* In proc. SIGCSE'04, (2004), pp. 26–30.
8. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
9. M. Joy, N. Griffiths, R. Boyatt, The BOSS Online Submission and Assessment System. *ACM J. Educ. Res. Comp.* **5**(3), 2005.
10. S. Benford, E. Burke, E. Foxley, N. Gutteridge and A.M. Zin. Ceilidh, A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, (1993), pp. 107–110.
11. C. Higgins, P. Symeonidis, A. Tsintsifas, The marking system for CourseMaster. In proc. of the *7th annual conference on Innovation and technology in computer science education*, (2002), pp. 46–50.

12. C. Douce, D. Livingstone, J. Orwell, S. Grindle, and J. Cobb, (2005). Automatic Assessment of Programming Assignments, *In proceedings of ALT-C* (2005).
13. C. Daly and J. M. Horgan. An automated learning system for Java programming, *IEEE Transact. Educ.* **47**(1), (2004), pp. 10–17.
14. S. H. Edwards, Improving student performance by evaluating how well students test their own programs. *J. Educ. Res. Comp.* **3**(3) (2003), pp. 1–24.
15. J. English, Experience with a computer-assisted formal programming examination. In *Proc. 7th annual conference on Innovation and technology in computer science education*, (2002), pp. 51–54.
16. D. Jackson and M. Usher, Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education*, (1997) pp. 335–339.
17. Malmi L., Korhonen A., Automatic Feedback and Resubmissions as Learning Aid, *Proceedings of the IEEE International Conference on Advanced Learning Technologies. ICALT'04.* (2004).
18. S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia, PILOT: An interactive tool for learning and grading. In *The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, (2000), pp. 139–143.
19. S. Hussmann, G. Covic and N. Patel, Effective Teaching and Learning in Engineering Education using Novel Web-based Tutorial and Assessment Tool for Advanced Electronics, *Int. J. Eng. Educ.* **20**(2), (2004), pp. 161–169.
20. M. Negahban, Results of Implementing a Computer-based Mechanics Readiness Program in Statics, *Int. J. Eng. Educ.* **16**(5), (2000), pp. 408–416.
21. A. Tartaglia and E. Tresso, An Automatic Evaluation System for Technical Education at the University Level, *IEEE Transact. Educ.* **45**(3), (2002), pp. 268–275.
22. A. Garcia-Beltran and R. Martinez, Web Assisted Self-assessment in Computer Programming Learning Using AulaWeb, *Int. J. Eng. Educ.*, **22**(5), (2006), pp. 1063–1069.
23. N. E. Fenton and S. L. Pfleeger, *Software metrics: a rigorous and practical approach*, PWS (1997).

APPENDIX A—DATA ANALYSIS

A.1 Course evaluation

Using variables presented in Table 9, the following null and alternative hypotheses are formulated:

H1_0: there is no difference in terms of Satisfaction in the traditional and new course modality.
H1_a: there is a difference in terms of Satisfaction in the traditional and new course modality.

Since the dependent variable is ordinal, to compare the student satisfaction obtained with the traditional approach vs. the new one, the non parametric Mann-Whitney test, with an alpha level of 5% is opted for. The results were cross checked by also applying the t-test. The results are summarized in Table 10.

The null hypothesis that there is no difference in terms of student satisfaction can be rejected. Statistically significant evidence proves that students like the new approach more.

A.2 Comparison Questionnaire

The answers to questions Q1 through Q4 can be encoded and treated as measures of the perceived learning support provided by an exam type in a specific course modality. The variables are summarized in the following Table 11.

Two null hypotheses are tested stating that no difference exists in terms of perceived usefulness of lab and exam procedure for learning of APA and OOP courses. In detail:

H2_0: for lab sessions Support in APA course is equal to support in OOP course
H3_0: for exam sessions Support in APA course is equal to support in OOP course
H4_0: overall the Support in APA course is equal to support in OOP course

Table 9. Definition of variables

Variable	Type	Description
Satisfaction	Likert scale encoded into and ordinal: 1 to 4 Dependente	Answer to the question ("are the labs useful and managed effectively?") being 4 a fully positive answer
MODE	Nominal: {Traditional, New} Independent	The course modality, either Traditional or new

Table 10. Student satisfaction: traditional vs. new approach

		Traditional	New	p-value	Test
Satisfaction	Median	3	4	0.0002	Mann-Whitney
	Mean	2.58	3.43	< 0.001	t-test

Table 11. Definition of variables

Variable	**Type**	**Description**
Support	Likert scale encoded into and ordinal: 1 to 5 Dependent	Learning support perceived in the programming session
Type	Nominal: {lab, exam} Independent	Type of session: either laboratory or exam
Course	Nominal: {APA, OOP} Independent	The course modality, either traditional paper based APA exam or the PC-based OOP exam

Table 12. Direct comparison contingency table

	PC based	Undecided	Paper based
Lab	37	4	4
Exam	35	4	6

To test the above hypotheses we use a paired two-tailed Mann-Whitney test because the data are not on an interval scale and not normally distributed.

All three null hypotheses (p values < 0.001) can be rejected.

The results from question Q5Q7 and Q6Q8 that can be summarized in the contingency table presented in Table 12.

Since the values are nominal, two Chi-Square tests are used on the two rows of the table. P-values are < 0.001 in both cases. The odds of a student preferring the PC-based approach versus the paper-based one are 9.2 (= 37/4) for the lab and 5.8 (= 35/6) for the exam.

Marco Torchiano received the M.Sc and Ph.D degrees in computer engineering from the Politecnico di Torino, Italy, where he is currently an assistant professor. He was a postdoctoral research fellow at the Norwegian University of Science and Technology (NTNU), Norway. He has published more than 50 research papers in international journals and conferences. He is a coauthor of the book *Software Development—Case Studies in Java* (Addison-Wesley) and is a coeditor of the book *Developing Services for the Wireless Internet* (Springer). His current research interests include empirical software engineering, OTS-based development and software engineering for mobile and wireless applications.

Maurizio Morisio received the Ph.D degree in software engineering and the M.Sc degree in electronic engineering from the Politecnico di Torino, Turin, Italy, where he is currently an associate professor. He spent two years working with the Experimental Software Engineering Group at the University of Maryland, College Park. His current research interests include experimental software engineering, service engineering, software reuse metrics and models, agile methodologies and commercial off-the-shelf processes and integration. He is a consultant for improving software production through technology and processes.