Automatic Assessment of Java Code[☆]

David Insa, Josep Silva*

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain

Abstract

Assessment is an integral part of education often used to evaluate students, but also to provide them with feedback. It is essential to ensure that assessment is fair, objective, and equally applied to all students. This holds, for instance, in multiple-choice tests, but, unfortunately, it is not ensured in the assessment of source code, which is still a manual and error-prone task. In this paper, we present JavAssess, a Java library with an API composed of around 200 methods to automatically inspect, test, mark, and correct Java code. It can be used to produce both black-box (based on output comparison) and white-box (based on the internal properties of the code) assessment tools. This means that it allows for marking the code even if it is only partially correct. We describe the library, how to use it, and we provide a complete example to automatically mark and correct a student's code. We also report the use of this system in a real university context to compare manual and automatic assessment in university courses. The study reports the average error in the marks produced by teachers when assessing source code manually, and it shows that the system automatically assesses around 50% of the work.

Keywords: Assessment, Java
2010 MSC: 97Q70, 97P40

1. Introduction

Assessment is a fundamental part of education, and it is useful both for students, who receive diagnostic feedback about their learning process, and for teachers, who can determine whether or not the goals of education are being met. In the psychology education area, it has been proved that most students often direct their efforts based on what is assessed

[☆]This work has been partially supported by MINECO/AEI/FEDER (EU) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), and by the Universitat Politècnica de València under grant PIME B18 and EICE HEGEA.

*Corresponding Author. Phone Number: (+34) 96 387 7007 (Ext. 73530)
Email addresses: `dinsa@dsic.upv.es` (David Insa), `jsilva@dsic.upv.es` (Josep Silva)
URL: `http://www.dsic.upv.es/~dinsa/` (David Insa), `http://www.dsic.upv.es/~jsilva/` (Josep Silva)

and how it affects the final course mark (see, e.g., [5], chapter 9). As a consequence, continuous assessment during a course can be used to direct and enhance the learning process. However, providing quality manual assessment for even a small class requires an important effort. When the size of the class grows, the amount of assessed work has to be limited or rationalized in some way.

Moreover, interiorizing all the assessment criteria for all possible situations is almost impossible, and, in practice, two teachers of the same subject very rarely apply the same assessment criteria in all cases. This is clearly unfair, because it means that a student's mark can depend on the teacher who assesses their solution, and not only on the student's solution itself. This happens, e.g., in the assessment of programming exercises. Often, there exist infinite possible solutions to the same problem because there can be variations in the programming style, the documentation, the functionality, the efficiency, the maintainability of the code, etc. Therefore, an assessment template can provide guidelines, but often, it cannot be exhaustive. This leaves room for interpretation.

1.1. Manual versus automatic assessment

We measured the errors made by teachers when assessing Java exercises. For that, based on the library presented in this paper, we implemented a tool able to automatically and precisely mark some specific properties of the code. A property is a requirement an exercise must fulfil (e.g., "a class must extend another class", "a field must be private", etc.). With this tool, we carried out an experiment in which we marked several Java exams of real university programming courses at Universitat Politècnica de València. Concretely, the experiment was performed in a second-year Java course whose curricula includes inheritance, abstract classes, interfaces, packages, polymorphism, etc., and whose exams consist in the development of a class model with around 10 classes. The experiment collected data from two academic years: 5 teachers (not ourselves) manually, as usual, marked three exams along the first year, and three exams along the second year. 535 students participated (381 the first year, and 154 the second year). Then, we marked the exams with our tool and compared the results. They are summarized in Figure 1.

In the table, each row represents a different exam. The meaning of the columns is the following: Column `#Students` contains the amount of assessed exams. Column `Average Mark` contains the average mark of each exam (this is shown for both the manual and the

		Automatic Assessment		Manual Assessment		Difference	
	#Students	Average Mark	Standard Deviation	Average Mark	Standard Deviation	%	Absolute
Exam 1	129	6,49	3,71	6,66	3,77	2,62%	0,27
Exam 2	129	3,88	3,47	3,99	3,26	2,84%	0,23
Exam 3	123	5,7	2,97	4,45	3,09	-21,93%	1,52
Exam 4	63	7,25	2,23	7,39	2,37	1,93%	0,49
Exam 5	31	5,24	3,36	5,31	3,29	1,34%	0,15
Exam 6	60	6,02	3,18	6,28	3,03	4,32%	0,29
Total/Average	535	5,76	3,15	5,68	3,14	5,83%	0,49

Figure 1: Comparison of automatic and manual assessment of real university Java exams

automatic assessment). The standard deviations of the marks are shown in column `Standard Deviation`. Finally, column `Difference` compares both average marks, where `%` shows the difference in the final mark between the manual and the automatic assessment. The total (absolute) error per exam is shown in column `Absolute`.

To the best of our knowledge, this is the first study that compares manual and automatic assessments of programming exercises. The study is large enough (2 subjects, 5 teachers, 535 students, 6 exams) to produce statistically valid results. The error shown in column `%` is the observable error made in the manual assessment, but it hides the real total amount of errors made by the teachers. To compute the total absolute error (0.49 points, in absolute value, out of 10) it is important to consider that errors made by teachers were 75% of the times positive (they benefitted the student) and 25% of the times negative (they harmed the student), thus many times they compensated themselves. This compensation happens both when marking one single exam, and also when computing the average of different exams. Therefore, the absolute error is computed as the average of the errors made in each exam being the errors an absolute value.

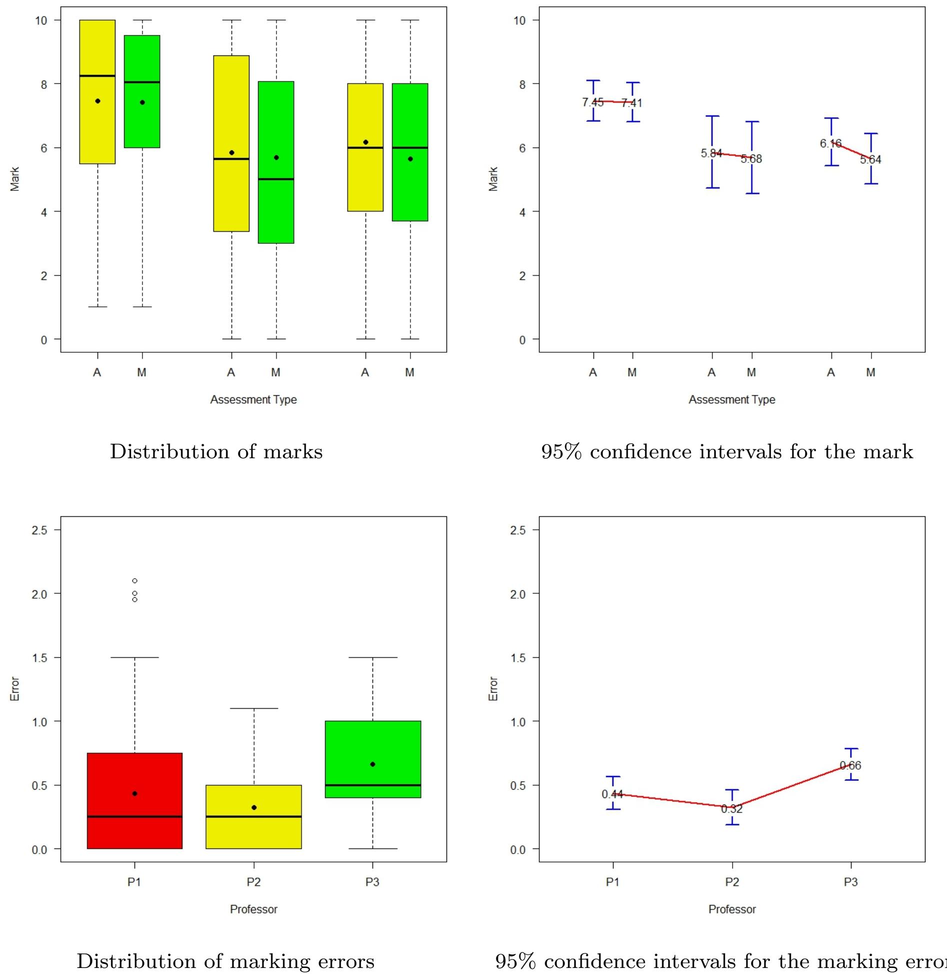
We studied with the teachers the causes of these marking errors. In almost all cases, they were due to teachers' mistakes in the manual assessment, and in a few cases they were due to small differences in the interpretation of the assessment criteria (e.g., unspecified details that were penalized with -0.1 the first time and with -0.2 the second time).

The errors made in the manual assessment were due to:

1. Wrong code introduced by the students in classes not involved in the exercise (and thus, not revised by the teacher and not penalized),
2. type errors not affecting the result,
3. incorrect use of interfaces,
4. code that is correct but it is marked as wrong because it is surrounded by wrong code,
5. messy code very difficult to understand even though it is correct,
6. introduction of dead code, or even
7. teachers' errors when marking the exercises. This happened massively in the third exam and produced the anomalous difference of 21.93%: One teacher wrongly marked one question for all students.

Figure 2 compares the assessment made by three professors with the same exams. We can observe a standard boxplot showing the distribution of marks in the top left corner. It shows the mean, Q1, Q2 (median), and Q3 quartiles for each professor. Here, A stands for automatic assessment and M stands for manual assessment. The average of the marks, centred inside a 95% confidence interval, is shown in the top right chart. What is more interesting for this study is revealed at the bottom of the figure. On the left, we see another boxplot presenting the distribution of the errors made by the three professors (P1, P2, and P3). The white circles in P1 are outliers. In this chart, it is made evident that the errors made by the professors are different, and distributed differently. In particular, in the chart on the right, we see that, e.g., professor P2 made an average marking error of 0.32 points (in

absolute value), while the marking error of professor P3 is over twice as much: 0.66 points (in absolute value).



Distribution of marks | 95% confidence intervals for the mark

Distribution of marking errors | 95% confidence intervals for the marking error

Figure 2: Comparison of the automatic and manual assessments made by three professors

In order to statistically validate the influence of professors on the marking errors, we performed an analysis of variance (ANOVA) and got the following result:

	Df	*Sum Sq*	*Mean Sq*	*F value*	*Pr(> F)*	
Professor	1	1.40	1.402	5.892	0.0164	*
Residuals	149	35.46	0.238			

With this result, and assuming a significance level of 0.05, we can reject the null hypothesis (which is: *professors do not have influence on the mark when marking*) because the probability value associated with the F Value, $Pr(> F) = 0.0164$, is below the significance level.

These empirical results show four important unacceptable problems of the manual assessment of programming exercises: (1) The average accumulated error is around 5%, (2) errors are distributed along a big interval, affecting each student in a different way, (3) different professors produce different errors and error distributions, making the assessment quality dependent on the professor, and (4) errors are sometimes positive and sometimes negative, producing an even more unfair assessment.

1.2. Advantages of automatic assessment

A wrong assessment is unfair for a student, who can be harmed in their mark, or more importantly, mislead in their instruction. For this reason, we advocate for automatic assessment (AA), which provides important advantages:

1. **Speed**. The assessment is much quicker, and a student gets feedback straight after the exam. In the manual assessment done in the experiment, the students made an exam and, two weeks later, they received their marks via email. Of course, they could then review the exam but, after two weeks, they hardly remembered the details of the exam. So, effectively, the exam was a summative assessment, but not formative. In contrast, with AA, straight after the exam, the students could have self-assessed their own exams, known their marks, and corrected their own errors, having available a self-corrected version of their own code.

2. **Fairness**. The assessment is fair. Same errors are evaluated equally, and teachers have no influence on the marking.

3. **Independence**. The assessment of one exam is not affected by the assessment of the previous exams. In the manual assessment, a very good (respectively very bad) exam can subjectively make a teacher to consider the next exam as bad (respectively good) in comparison.

Unfortunately, AA of programming code is far from being a reality. Firstly, because there are no tools or infrastructure to automatically assess source code. Concretely, the ability of assessing source code requires advanced language features such as introspection (i.e., the ability to inspect the structure/state of a program at runtime) and intercession (i.e., the ability to change the structure/state or the behaviour/meaning of a program at runtime). Unfortunately, most languages lack of these features, or, if they implement them, they do it in a very restrictive way. In this work, we overcome these limitations and we provide the first library specific to build assessing tools and self-assessing exercises.

1.3. Contributions

We envisage a future in which students have available repositories of classified self-assessing exercises prepared to learn different aspects of programming (inheritance, interfaces, abstract classes, generic classes, etc.). A self-assessing exercise is an artifact that contains a statement of a problem, the resources needed to solve it (images, libraries, etc.), a solution of the problem, test cases, etc. But the most important thing it contains is an

assessment template, which is a source code that analyzes the student's solution, uses the test cases to validate the output of the student's code against the output of the teacher's solution, generates an assessment report, etc.

The assessment template is the core of a self-assessing exercise. By definition, it is a meta-program that manipulates the student's solution code, the teacher's solution code, and all the other components.

This is actually possible if we enhance the meta-programming capabilities of the language and provide an API to manipulate and assess programs. In this work, we have done it for Java. We present the first Java library with all the necessary infrastructure to assess programs by automatically validating any properties such as the use of interfaces, the existence of a particular class hierarchy, the specification of one concrete method or field, etc.

Example 1.1. *In an exam, a student implements the following code:*

Student's solution

```
class Student {
    int studCardId;

    Student(int studCardId) {
        this.studCardId = studCardId;
    }

    public int getStudCardId() {
        return this.studCardId;
    }
}
```

```
class PhDStudent {
    int studCardId;
    int scholarship;

    PhDStudent(int studCardId, int scholarship) {
        this.studCardId = studCardId;
        this.scholarship = scholarship;
    }

    public int getStudCardId() {
        return this.studCardId;
    }

    public int getScholarship() {
        return this.scholarship;
    }
}
```

An expected solution is shown below:
Teacher's solution

```
class Student implements Person {
    protected int studCardId;

    Student(int studCardId) {
        this.studCardId = studCardId;
    }

    public int getStudCardId() {
        return this.studCardId;
    }
}
```

```
class PhDStudent extends Student {
    private int scholarship;

    PhDStudent(int studCardId, int scholarship) {
        super(studCardId);
        this.scholarship = scholarship;
    }

    public int getScholarship() {
        return this.scholarship;
    }
}
```

Observe that both solutions have the same fields and methods, and thus, for most tests, they will always produce the same output. However, the teacher's solution is more maintainable and reuses code via inheritance. Moreover, it makes use of an interface, so other objects in the system know a priori that all `Student` *objects can behave as a* `Person`*.*

The student forgot to use the interface and also the inheritance, and he/she did a bad use of modifiers (protected, private, etc.). To the best of our knowledge, there does not exist

a system able to automatically identify these problems and correct them.

In the following sections, we show how this code can be automatically marked and corrected with our library. In summary, the main contributions of our work are the following:

1. An empirical evaluation and statistical analysis of the assessment errors produced by professors.

2. A Java library for the automatic marking and assessment of exercises.

3. A collection of routines to automatize assessment tasks (e.g., automatically loading classes at runtime when the source code changes).

1.4. Outline of the paper

The rest of the paper analyzes the problems that must be tackled to produce AA of source code, proposes solutions to these problems, and presents JavAssess, a library that implements the proposed solutions. First, in Section 2, we describe the related work, with special emphasis on the current systems for AA of source code. Then, in Section 3, we discuss the requirements and problems to overcome. In Section 4, we present JavAssess and describe its structure and how to use it with a complete example. In Section 5, we describe our experience using an AA system implemented with JavAssess that is being used in real university courses. Finally, the conclusions are discussed in Section 6.

2. Related work

According to Pears et al. [19], tools that support teaching programming can be classified as follows:

Programming support tools. They comprise a collection of tools that enhance the programming experience. They assist the programmer and help them to understand their programs. Examples of these tools are programming environments, debuggers, and profilers, among others. See, e.g., BlueJ [3].

Microworlds. These are systems specifically designed to teach programming. They include tools and interfaces that can be used to learn different programming concepts. They are often used by students as a training system. See, e.g., Karel [18], and Alice [6].

Visualization tools. These systems allow us to animate algorithms, so that the execution can be visually inspected step by step. They are a useful resource for program comprehension. See, e.g., Animal [21], Jeliot [16], and Tango [25].

AA tools. These tools are designed to assist teachers in the assessment process. See, e.g., TRAKLA2 [13], WEBCAT [8], and BOSS [11].

In this paper, we focus on the area of AA tools. Most of the work in this area has been done for the multiple-choice test systems. In fact, AA of multiple-choice tests has been a hot topic for a long time [10]. The good results obtained in this area generated a great interest in university courses with a large number of students. For this reason, it has been systematically implemented in most universities. As a result, many advances have been done in this area, and, nowadays, there exist many different ways for a teacher to define tests, resubmission policies, security issues, improving the scanning process, allowing for recognition of students' annotations (e.g., permitting to alter an answer by annotating the "error" circle and handwriting the letter of the correct answer next to the appropriate row), or enabling automated reading of a limited set of handwritten answers, thus minimizing the need for a human intervention. Some remarkable systems of this kind are [26, 9].

A second kind of AA tools is based on *visual algorithm simulation exercises* [14]. These tools use advanced GUIs that control all possible (finite) student's actions regarding a particular problem (e.g., sorting an array). In visual algorithm simulation exercises, a learner directly manipulates the visual representation of the underlying data structures to which the algorithm is applied. The learner manipulates these real data structures through GUI operations with the purpose of performing the same changes on the data structures that the real algorithm would do (e.g., a student can simulate the steps of Quicksort using the GUI, and the system can assess every single movement using the real Quicksort algorithm) [14].

Our approach falls within a third kind of AA tool that has been less investigated: AA of source code. In this area, most approaches use black-box techniques, i.e., based on output comparison (see, e.g., [20, 7], and the five surveys [2, 1, 15, 10, 24]). The general idea is to compile the student's code, execute it with a predetermined set of inputs, and compare the outputs with the expected results. Therefore, the student's source code is not really analyzed, only its output. The most advanced systems [28, 12, 4] also use random test case generation and some sort of model (often the correct algorithm) to compare the results. Some of these systems are language-independent. This is possible because they take the compiler as an argument, and they just compare the results. Of course, the quality of the assessment in these systems is strongly dependent on the quality of the test cases used.

Unfortunately, there are very few white-box approaches, i.e., that perform code analysis to mark exercises. Two exceptions are [17] and [29]. In both approaches, the idea is to measure the similarity of a student's code with the pool of known solutions. Similarity is computed with a graph representation of the code. A systematic review of the most important AA tools can be found in [24]. Other previous reviews are [2, 15, 1, 10].

Our approach is a library that allows for the implementation of white-box systems. It goes beyond output comparison, and allows a teacher to assess any internal properties of the code. Moreover, it is not limited to functional properties. It can also assess usability properties, efficiency, programming style, etc. Another library that aids the user in testing their programs is LIFT [23], which focusses on allowing the user to create test cases that interact with the GUI (e.g., "click this button", "drag the mouse from here to here"). In contrast, the part of our tool that focusses on testing allows for executing test cases, so both libraries can be used in a complementary way. Besides test case execution, our library also allows for property checking and code correction, which is beyond the capabilities of LIFT.

In contrast to many other approaches [10], our library has not been designed for a specific Java programming course useful for a single teacher, but a library that can be widely used, configured and augmented to design any Java exercise or exam and to automatically assess batteries of exercises.

3. Requirements, problems, and solutions for an AA system

We are not aware of any other library to specify properties that can be automatically validated, corrected, and marked. The reason is the difficulty in implementing such a library with the current programming resources provided by languages.

An assessment system (AS) needs advanced meta-programming features such as:

1. Source code manipulation. The AS needs a mechanism to reify the source code of the exercise submitted by the student, in such a way that we can manipulate the code, access its structure, reimplement the statements, etc. This effectively means that we need mechanisms to:
   - (a) Parse the source code.
   - (b) Build an abstract syntax tree (AST) representing every detail of the program. It is desirable that ASTs retain comments and other details of the source code layout such as column numbers, literal values, etc.
   - (c) Traverse and analyze the code using the AST.
   - (d) Transform the code (usually by modifying the AST that represents the parsed code).
   - (e) Regenerate source code (including layout and comments) from the modified AST.
2. Reflection. Reflection is commonly used by programs that require the ability to examine or modify the runtime behaviour of applications. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language, because a bad use of reflection can produce performance overhead, security violations, and undesired exposure of the internal properties/state/behaviour of the program. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations that would otherwise be impossible. There are two different kinds of reflection, both of which are necessary in ASs:
   - (a) Introspection. The AS must be able to inspect the structure/state of the program that is being assessed. It is needed for marking the student's code.
   - (b) Intercession. The AS must be able to change the structure/state/behaviour/meaning of the program. It is needed for correcting the student's code.

In the specific case of Java, even though it offers one of the most advanced reflection mechanisms, some of the reflection aspects we need in our library are not provided. For

instance, in order to assess a method that should use two local variables, we need to consult the variables defined inside this method. Unfortunately, this is beyond the current capabilities of the Java's reflection feature.

The limits of Java's reflection are:

1. Introspection. The introspection feature provided by Java is limited to the method level. This means that if you have an object reference, you can get its class (i.e., `object.getClass()`). If you have a class, you can know its full description, for example, its package name, its access modifiers, all field types, the methods defined in the class, their return types, access modifiers, parameter types, parameter names, etc. But you cannot access the code inside its methods. This means that it is not possible to know, e.g., the local variables or the number of loops used inside a method.

2. Intercession. The intercession feature provided by Java lets one create an instance of a class whose name is unknown until runtime, invoke methods on such instances, and get/set its fields. However, Java does not allow one to modify objects at runtime. For example, we cannot add a new field or a method to an object at runtime because all fields of an object are always determined at compile time ([22] Chapter 3). Examples of behavioural intercession are the ability to change the method execution at runtime and add a new method to a class at runtime. Java does not provide any of these intercession facilities. That is, we cannot modify the code of a method at runtime to change its behaviour; neither can we add a new method to a class at runtime.

There is another important problem that we must face: if we interleave marking and correcting operations in an exercise, this means that we can have different versions of the same source code (the original version, and one new version for each correction done in the code). Hence, we need a mechanism to dynamically (at assessment time) compile the different versions of the code, and load the appropriate classes into the JVM, so that introspection operations are done on the desired code (e.g., on the last modified version of the code).

4. The assessment library JavAssess

We have implemented a library that provides solutions to the described problems. Our assessment library is not an assessment system, but a resource to create assessment systems. It can be imported normally as any other Java library. JavAssess is a Java library with heavy use of reflection that provides an API composed of around 200 methods that can be used to inspect, analyze and change Java source code. To overcome the intercession features that are missing in Java, we use an external Java library called JavaParser [27], which is able to produce and manipulate ASTs obtained from the source code. On top of JavaParser, we have implemented a part of our library composed of a collection of methods that are specific for the assessment of Java code. JavAssess is under continuous development. The current complete API of JavAssess is available at:

`http://www.dsic.upv.es/~jsilva/JavAssess/`

4.1. Structure of JavAssess

JavAssess is composed of four different modules, three of which can work in an isolated way. The three independent modules are classes that are located in the *codeassess.javassessment* package, and are called `introspector`, `intercessor`, and `tester`. The fourth extra module internally uses the other three to exploit their functionality while allowing for combining their effects. It is implemented in the *codeassess* package by the *CodeAssess* class and it requires an instance of a class that should be written by the programmer. The programmer's class must extend the *Assessor* class located in the *codeassess.javassessment* package and, by doing so, it automatically contains the three previous modules. These modules internally make all the abstraction, which is made transparent to the user. For instance, class *Introspector* provides several methods to query and manipulate the meta information of the class in order to check properties of the code.

Example 4.1. *Some methods of the library, useful to introspect the student's code, follow:*

- ```
  public Class<?> getSuperclass(Class<?> clazz)
  ```
  Returns the superclass of a given class.

- ```
  public boolean checkType(Field field, TypeVariable<?> genericType)
  ```
  Checks whether the type of a given field is the generic type given as second argument.

- ```
  public boolean checkGenericType(Field field, int[] indices)
  ```
  Checks whether a given field contains a generic type in a specified position (e.g., in Map<T, Map<S, R>> the indices [1, 0] stand for the generic type S).

- ```
  public Method getDeclaredMethod(Class<?> clazz, String methodName,
                Class<?>[] paramClasses, Class<?> returnClass, boolean allowWrappers)
  ```
  Returns the method with the given name and parameters and return types. It can allow a wrapper as any parameter or returned type instead of the primitive ones.

- ```
  public Class<?> getReturnClass(Method method)
  ```
  Returns the class of the return type of a method.

Some methods of the library, useful to intercede the student's code, follow:

- ```
  public void setSuperclass(Class<?> clazz, Class<?> superclass)
  ```
  Modifies the student's class clazz[1], *replacing its super class with the specified one.*

- ```
  public void addImplement(Class<?> clazz, Class<?> interfaze)
  ```

[1]Note the careful spelling to avoid name classes with the reserved words of Java.
  ```

Modifies the student's class clazz*, adding the* interfaze *interface to the interfaces it implements.*

- `public void copyMethod(Class<?> clazz, String methodName, Class<?>[] paramClasses)`

  Copies the method with the given name and parameters contained in the clazz *class from the teacher's solution to the student's solution.*

Some methods of the library, useful to test the student's code, follow:

- `public double executeTestCases(String className)`

  Executes all the test cases that are stored in the specified class. All test cases are supposed to return a boolean indicating whether the test case has been passed. A double is returned representing the percentage of test cases passed.

- `public Object executeTestCase(String className, String methodName, Object... args)`

  Executes the test case that is in the specified class and method with the specified arguments.

Observe that all functions are generic, so that they can work with any type.

4.2. How to use JavAssess to mark, correct and test code

With JavAssess, we can produce code able to automatically assess an exercise, or even self-assessing exercises. Hereafter, we show three examples of how the library is used to mark, assess, and test Java exercises.

4.2.1. Automatically marking with JavAssess

Given the teacher's solution code in Example 1.1, we could specify the following marking criteria:

Criterion 1: Class `Student` only has one field and it is of type `int`: 3 points

Criterion 2: Class `Student` only implements one interface (`Person`): 3 points

Criterion 3: Class `PhDStudent` extends class `Student`: 4 points

The following code checks the three properties (methods `introspector.*` are provided by the library and have the obvious meaning):

```
import codeassess.javassessment.Introspector;

public class Example1 {
    public double assess() {
        String[] projectPaths = { "/path/project/" };
        Introspector introspector = new Introspector(projectPaths);
        Class<?> studentClass = introspector.getClass("Student");
        Class<?> phdStudentClass = introspector.getClass("PhDStudent");
        Class<?> personClass = introspector.getClass("Person");
```

```
        // Criterion 1
        boolean oneField = introspector.getDeclaredFieldCount(studentClass) == 1;
        Field[] fields = introspector.getDeclaredFields(studentClass);
        boolean isFieldInt = oneField && introspector.checkClass(fields[0], int.class, true);

        // Criterion 2
        boolean oneInterface = introspector.getInterfaceCount(studentClass) == 1;
        boolean implementsPerson = introspector.checkImplements(studentClass, personClass);

        // Criterion 3
        boolean extendsStudent = introspector.checkSuperclass(phdStudentClass, studentClass);
```

By adding the following code, we can automatically mark the exam:

```
        // Mark
        double mark = 0;
        if (oneField) mark += 1.5;
        if (isFieldInt) mark += 1.5;
        if (oneInterface && implementsPerson) mark += 3;
        if (extendsStudent) mark += 4;
        return mark;
    }
}
```

4.2.2. Automatically correcting code with JavAssess

Correcting the code of an exercise requires class `Intercessor`. For instance, if we want to correct the first line in Example 1.1, we need to add `implements Person` to the student's code. In order to correct the exercise code to fulfil all the criteria, we can use the following code:

```
import codeassess.javassessment.Introspector;
import codeassess.javassessment.Intercessor;
import java.lang.reflect.Modifier;

public class Example2 {
    public double assess() {
        String[] projectPaths = { "/path/project/" };
        String[] solutionPaths = { "/path/solution/" };
        Introspector introspector = new Introspector(projectPaths);
        Intercessor intercessor = new Intercessor(projectPaths, solutionPaths);
        Class<?> studentClass = introspector.getClass("Student");
        Class<?> phdStudentClass = introspector.getClass("PhDStudent");
        Class<?> personClass = introspector.getClass("Person");

        // Criterion 1
        intercessor.removeFields(studentClass);
        intercessor.addField(studentClass, Modifier.PROTECTED, int.class, "studCardId");

        // Criterion 2
        intercessor.removeImplements(studentClass);
        intercessor.addImplement(studentClass, personClass);

        // Criterion 3
        intercessor.setSuperclass(phdStudentClass, studentClass);
    }
}
```

4.2.3. Automatically testing code with JavAssess

If we want to test whether the methods implemented by the student behave as expected, we can create the following test cases:

```java
public TestCases {
    public boolean testStudCardId(int studCardId) {
        Student student = new Student(studCardId);
        return student.getStudCardId() == studCardId;
    }
    public boolean testScholarship(int scholarship) {
        PhDStudent pdhStudent = new PhDStudent(0, scholarship);
        return pdhStudent.getScholarship() == scholarship;
    }
}
```

To execute these tests, we can use the `Tester` class. An example of how to use this class follows:

```java
import codeassess.javassessment.Tester;

public Example3 {
    public boolean test() {
        String[] projectPaths = { "/path/project/" };
        String[] testCasesPaths = { "/path/testCases/", "/path/testCases2/" };
        Tester tester = new Tester(projectPaths, testCasesPaths);

        boolean result1 = tester.executeTestCase("TestCases", "testStudCardId", 1000);
        boolean result2 = tester.executeTestCase("TestCases", "testStudCardId", 1004);
        boolean result3 = tester.executeTestCase("TestCases", "testStudCardId", 1408);

        if (!result1 || !result2 || !result3)
            return false;

        boolean result4 = tester.executeTestCase("TestCases", "testScholarship", 7530);
        boolean result5 = tester.executeTestCase("TestCases", "testScholarship", 2100);
        boolean result6 = tester.executeTestCase("TestCases", "testScholarship", 4000);

        if (!result4 || !result5 || !result6)
            return false;
        return true;
    }
}
```

One interesting property of the `Tester` class is that it can use the teacher's solution as an oracle. This means that we can automatically produce the test cases, i.e., we first execute the test cases against the teacher's solution, and then against the student's solution to check whether they produce the same outputs. With this ability, the testing phase can be completely automatic. In fact, we have implemented a test case generator that automatically produces (thousands of) random inputs for each method in the teacher's code (using introspection), and then, it automatically compares the corresponding outputs with the student's solution. With this infrastructure, any method can be automatically and massively tested with the generated test cases.

4.3. How to use JavAssess inside an assessment tool

In Sections 4.2.1, 4.2.2, and 4.2.3, we showed how to use the *Introspector*, *Intercessor*, and *Tester* classes to mark, correct, and test the students' classes, respectively. However, the assessment code implemented there is made ad hoc for the code in Example 1.1. Therefore, the assessment of the different properties is mixed together with the code needed to compute the final mark. This is a bad programming practice that determines problems during maintenance. A good design, should separate the code related to the whole exam

(e.g., producing the final mark) from the code related to each individual property, which in turn should be implemented separately. Moreover, it should separate the code associated with the assessment of the exam from the code associated with the information needed to assess the exam (e.g., the assessment criteria).

Our library already provides mechanisms to produce a good design that ensures cohesive and maintainable code. First, all the assessment criteria should be grouped inside a class extending class *Assessor*. We often call this class *Template* because it acts as an assessment template. It indicates how the assessment tool should behave to assess a student's solution. To avoid the programmer having to declare and initialize the *Introspector*, *Intercessor*, and *Tester* classes, the *Assessor* class already does it. An example of a template class follows:

```
import codeassess.javassessment.Assessor;

public Template extends Assessor {
    public List<Object[]> getCriteria() {
        List<Object[]> criteria = new ArrayList<Object[]>();
        criteria.add(new Object[]{ "criterion1", 3.0 });
        criteria.add(new Object[]{ "criterion2", 3.0 });
        criteria.add(new Object[]{ "criterion3", 4.0 });
        return criteria;
    }

    public double criterion1() {
        Class<?> studentClass = introspector.getClass("Student");
        boolean oneField = introspector.getDeclaredFieldCount(studentClass) == 1;
        Field[] fields = introspector.getDeclaredFields(studentClass);
        boolean isFieldInt = oneField && introspector.checkClass(fields[0], int.class, true);

        if (oneField && isFieldInt) return 3.0;

        intercessor.removeFields(studentClass);
        intercessor.addField(studentClass, Modifier.PROTECTED, int.class, "studCardId");
        super.refresh();

        if (oneField || isFieldInt) return 1.5;

        return 0.0;
    }
    public double criterion2() {
        Class<?> studentClass = introspector.getClass("Student");
        Class<?> personClass = introspector.getClass("Person");
        boolean oneInterface = introspector.getInterfaceCount(studentClass) == 1;
        boolean implementsPerson = introspector.checkImplements(studentClass, personClass);

        if (oneInterface && implementsPerson) return 3.0;

        intercessor.removeImplements(studentClass);
        intercessor.addImplement(studentClass, personClass);
        super.refresh();

        return 0.0;
    }
    public double criterion3() {
        Class<?> studentClass = introspector.getClass("Student");
        Class<?> phdStudentClass = introspector.getClass("PhDStudent");
        boolean extendsStudent = introspector.checkSuperclass(phdStudentClass, studentClass);

        if (extendsStudent) return 4.0;

        intercessor.setSuperclass(phdStudentClass, studentClass);
        super.refresh();
```

```
        return 0.0;
    }
}
```

This code contains all the information needed to automatically mark and correct the source code of Example 1.1. Note that each criterion is marked and corrected separately (using a different method), so the assessment tool only has to invoke these methods and, for each criterion, the template would automatically return its mark. In addition, an extra method *getCriteria* has been defined to indicate the criteria the exercise contains together with their maximum marks and additional info, if need be. Finally, it is important to note the use of the *refresh* method. Each time the intercessor object has finished to correct the code, the *refresh* function must be called to ensure that the (new) corrected code is the one used in the following assessing actions.

JavAssess is designed in such a way that different templates can be used to assess different exercises. The code needed to process a template and use it against a student's code (e.g., to mark it, or to correct it) is implemented in a separate class called *CodeAssess*. This could produce a situation where two professors used the same name (e.g., *InheritanceExercise*) for their templates. In this case, if we try to load a second class with a name already used, Java would detect that a class with the same name has already been loaded, and would return it instead of loading the new one. This situation can be solved by using different class loaders. *JavAssess* already implements this loading system and abstracts away the problem from the user. It can be used as follows:

```
import codeassess.CodeAssess;

public class Exemple4 {
    public double assess() {
        String templatePath = "/path/";
        String templateName = "InheritanceExercise.java"
        String[] libraries = { "/path/library1.jar", "/path/library2.jar" };
        String[] projectPaths = { "/path/project/" };
        String[] testCasesPaths = { "/path/testCases/", "/path/testCases2/" };
        String[] solutionPaths = { "/path/solution/" };
        CodeAssess codeAssess = new CodeAssess(templatePath, templateName,
                libraries, projectPaths, testCasesPaths, solutionPaths);

        List<Object[]> criteria = (List<Object[]>) codeAssess.execute("getCriteria");
        double finalMark = 0.0;

        for (Object[] criterion : criteria) {
            String criterionName = (String) criterion[0];
            Object[] arguments = { };
            finalMark += (Double) codeAssess.execute(criterionName, arguments);
        }

        return finalMark;
    }
}
```

Note that the *CodeAssess* class also provides the *execute* method, which invokes the specified method of the template with the given arguments.

5. An implementation of an AA system with JavAssess

We have implemented an assessment system for Java called ASys with JavAssess (see the website of the project: `http://www.dsic.upv.es/~jsilva/ASys/`, and the website of the tool: `http://asys.dsic.upv.es/`). Thanks to the advanced reflection capabilities of JavAssess, besides output comparison, this system can compile, execute, analyze, and evaluate the students' code statically and dynamically, analyzing the efficiency, the complexity, the structure, and the programming style of a given code. Furthermore, it can produce sophisticated templates able to check a student's programming ability (besides its coding ability).[2] For instance, the system can automatically check whether a student's solution:

1. has a given cyclomatic complexity to solve a problem;
2. makes a correct use of inheritance (in terms of programming, not coding), i.e., it can check whether the hierarchy of classes is correct (which contrast to checking whether the hierarchy is a specific one), and behaves correctly when the system is augmented with new classes or extended in other ways;
3. uses the right classes and interfaces (this includes accepting new classes invented by the student as correct, because they behave as specified, even though they are different from the teacher's solution);
4. correctly handles, e.g., polymorphism to solve a problem (no matter how it is implemented), because the system can inject the student's code into a different hierarchy of classes and check whether the objects still work due to the use of polymorphic types;
5. implements the right number of (nested) loops to solve a problem;
6. after some extent, belongs to the class of correct programs, which contrasts to checking whether this program is equivalent to a specified one (the teacher's solution).

This system is currently being used in two university programming courses with a double aim: (1) students use the tool to self-assess their exercises, and (2) teachers use the tool to automatically assess the final exam.

In Figure 3, we show the results obtained from a poll realized with 164 students. We also conducted interviews with both the students and the professors. In general, the students found the tool very useful to self-evaluate their learning. They defined it as a learning resource that can substitute the teacher in many situations, and this saves time to them, and also to the professor, because they asked considerably less questions. We were a bit surprised because only 45% considered that the tool helps them to pass the subject (see the last bars in Figure 3). In the interview, most of them clarified that they interpreted

[2] This distinction is important to difference when the system checks whether students have learned to use the right language constructs to express certain programming patterns, and when they actually decide to use the right programming patterns.

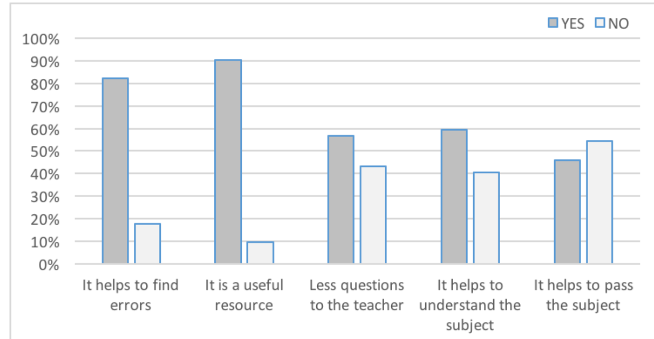


Figure 3: Evaluation of ASys by 164 students

the question as whether the tool “alone” is enough to pass the subject. They answered “no” because they considered that the teacher is still essential to pass the subject. The professors found the tool useful for three reasons: First, because it saves a lot of time in the assessment. Second, because it increases the students’s autonomy, and thus they received many less questions about how to solve the exercises. And third, because it increases the assessment quality, not only because it is exhaustive, also because it is completely fair and equally applied to all students.

It is important to remark that producing self-assessing exercises has a programming cost. In our tool, for introductory courses, the time needed to create a self-assessing exam is less than one hour. Note that the exercises in these exams are usually formed by one single class (e.g., adding attributes, completing the code of a method, etc.). For this kind of exams, ASys can make the assessment in a fully automatic way. In contrast, as the complexity of the exams grows up, the effort required to build a self-assessing exam is increased. Producing a self-assessing intermediate-level exam from scratch of around 10 classes can take around 4 hours. In practice, we rarely start from scratch, because many parts can be reused.

In complex exams, we do not use the automatic assessment mode, but the semi-automatic one: (i) the system automatically assesses the exercises until an error that cannot be automatically corrected is found (or because the professor specified in the assessment template to stop when this kind of error is found). Then, (ii) the professor is prompted with the student’s source code, the problem found, and the source code of the solution. (iii) The final mark is decided by the professor who can also modify the code and add comments to a report for the student.

One interesting lesson learned from the use of AA systems is that, even if a completely manual assessment is made, AA tools can help in the assessment. For instance, they can preprocess the exams and order them to enhance the assessment. As an example, our tool can automatically analyze the exams and order their questions according to the errors made. In this way, those exercises with the same error can be assessed together (thus having in mind the same problem and the same assessing criteria when they are corrected). This speeds up the assessment, and reduces the variability introduced by professors when the same error is marked.

6. Conclusions

The manual assessment of source code is a time-consuming and error prone task. In the same way that teachers make mistakes when programming, they also make them when they assess a program. This fact has been studied in the first part of this article with an experiment performed along two academic years in a real university context. The main result of the experiment is that manual assessment of programming code is imprecise and unfair (due to marking errors and error omissions) when the assessed code is not trivial.

Our experiments reveal that teachers made (as an average) an error of 5% and that this 5% can be avoided by automated tools. We could debate about whether this 5% is acceptable in the educational system, but this is not the point. The point is whether there is room for improvement in the educational system. A 5% discrepancy may lead to a situation where, for a code that deserves 5 out of 10 (being 10 the maximum mark and 5 the threshold to pass the exam), a student can be given 4.5 out of 10 (he/she does not pass the exam) whereas another student can be given 5.5 out of 10 (he/she does pass the exam). It is our responsibility as teachers to produce a marking as much fair and as much precise as possible, and this implies to make all efforts to produce and use assistance tools that improve our work.

White-box techniques have been historically ignored in assessment systems. Our library JavAssess makes a step forward in the design and development of Java assessment systems allowing for combining both black-box and white-box techniques. For this, it makes use of reflection together with other meta-programming features that allow us to manipulate the student's source code at runtime. The library contains three independent modules, called `Introspector`, `Intercessor`, and `Tester`; and a forth one, called `CodeAssess`, that allows for using them all together. The `introspector` module is used to analyze a student's source code in order to check desirable properties, and thus mark the exercise. The `intercessor` module can be used to correct the student's source code. Finally, the `tester` module allows for checking whether the behaviour of the student's code matches the teacher's one. The library contains over 200 methods that implement the four modules and is in continuous development.

Tool assistance can help not only to make part of the work automatically, but also to improve the quality and the fairness of the assessment. The library presented here provides useful resources to automatically identify errors, mark them, and correct them. It has been integrated in a real assessment system called ASys that has been used in the last two years in programming courses. The integration in a real assessment system has proved the usefulness of the library in practice. Currently, it is being adapted to other programming languages such as Haskell and Python.

References

[1] Abd Rahman, K., Jan Nordin, M., dec 2007. A review on the static analysis approach in the automated programming assessment systems. In: National Conference on Programming 07.

[2] Ala-Mutka, K. M., feb 2005. A survey of automated assessment approaches for programming assignments. Computer Science Education 15 (2), 83–102.

[3] Barnes, D. J., Kolling, M., April 2006. Objects First with Java - A Practical Introduction using BlueJ. Prentice Hall / Pearson Education.

[4] Beierle, C., Kulas, M., Widera, M., 2003. Automatic analysis of programming assignments. In: Proceedings of the 1. E-Learning Fachtagung Informatik (DeLFI'03). Verlang, pp. 144–153.

[5] Biggs, J., Tang, C., 2007. Teaching for Quality Learning at University : What the Student Does (3rd Edition). Society for Research Into Higher Education. Open University Press.

[6] Cooper, S., Dann, W., Pausch, R., may 2000. Alice: A 3-D tool for introductory programming concepts. Journal of Computing Sciences in Colleges 15 (5), 107–116.

[7] Denny, P., Luxton-Reilly, A., Tempero, E., Hendrickx, J., 2011. CodeWrite: Supporting student-driven practice of Java. In: Proceedings of the 42nd ACM technical symposium on Computer science education. ACM, New York, NY, USA, pp. 471–476.

[8] Edwards, S. H., Pérez-Quiñones, M., aug 2008. Web-CAT: Automatically grading programming assignments. ACM SIGCSE Bulletin 40 (3), 328–328.

[9] Hendriks, R., 2012. Automatic exam correction. Master's thesis, Universiteit Van Amsterdam.

[10] Ihantola, P., Ahoniemi, T., Karavirta, V., Seppala, O., 2010. Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research. ACM, New York, NY, USA, pp. 86–93.

[11] Joy, M., Griffiths, N., Boyatt, R., sept 2005. The BOSS online submission and assessment system. ACM Journal on Educational Resources in Computing 5 (3).

[12] Kitaya, H., Inoue, U., apr 2014. An online automated scoring system for Java programming assignments. International Journal of Information and Education Technology 6 (4), 275–279.

[13] Korhonen, A., Malmi, L., Silvasti, P., 2003. TRAKLA2: A framework for automatically assessed visual algorithm simulation exercises. In: Proceedings of the Third Annual Baltic Conference on Computer Science Education. pp. 48–56.

[14] Laakso, M.-J., Salakoski, T., Korhonen, A., Malmi, L., 2004. Automatic assessment of exercises for algorithms and data structures - a case study with TRAKLA2. In: Proceedings of Kolin Kolistelut/Koli Calling - Fourth Finnish/Baltic Sea Conference on Computer Science Education. pp. 28–36.

[15] Liang, Y., Liu, Q., Xu, J., Wang, D., dec 2009. The recent development of automated programming assessment. In: International Conference on Computational Intelligence and Software Engineering (CiSE 2009). IEEE, pp. 1–5.

[16] Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M., 2004. Visualizing programs with Jeliot 3. In: Proceedings of the Working Conference on Advanced Visual Interfaces. AVI '04. ACM, New York, NY, USA, pp. 373–376.

[17] Naudé, K. A., Greyling, J. H., Vogts, D., feb 2010. Marking student programs using graph similarity. Computers & Education 54 (2), 545–561.

[18] Pattis, R. E., 1994. Karel the Robot: A Gentle Introduction to the Art of Programming. John Wiley & Sons.

[19] Pears, A., Seidman, S., Eney, C., Kinnunen, P., Malmi, L., dec 2005. Constructing a core literature for computing education research. ACM SIGCSE Bulletin 37 (4), 152–161.

[20] Prados, F., Boada, I., Soler, J., Poch, J., 2005. Automatic generation and correction of technical exercises. In: International Conference on Engineering and Computer Education (ICECE 2005).

[21] Rosling, G., Freisleben, B., 2002. ANIMAL: A system for supporting multiple roles in algorithm animation. Journal of Visual Languages and Computing 13 (2), 341–354.

[22] Sharan, K., 2014. Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams. Expert's voice in Java. Apress.
URL `https://books.google.es/books?id=TnU_BAAAQBAJ`

[23] Snyder, J., Edwards, S. H., Pérez-Quiñones, M. A., 2011. Lift: Taking gui unit testing to new heights. In: Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, pp. 643–648.
URL `http://doi.acm.org/10.1145/1953163.1953343`

[24] Souza, D., Felizardo, K., Barbosa, E., 2016. A systematic literature review of assessment tools for pro-

gramming assignments. In: Proceedings of the 2016 International Conference on Software Engineering Education and Training (CSEET). IEEE, pp. 147–156.

[25] Stasko, J. T., sept 1990. TANGO: A framework and system for algorithm animation. IEEE Computer 23 (9), 27–39.

[26] Supic, M., Brkic, K., Hrkac, T., Mihajlovic, Z., Kalafatic, Z., 2014. Automatic recognition of handwritten corrections for multiple-choice exam answer sheets. In: International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). pp. 1136–1141.

[27] Tomassetti, F., Smith, N., Maximilien, C., Kirsch, S., 2017. Java Parser. Available at: `https://github.com/javaparser/javaparser`.

[28] Tung, S.-H., Lin, T.-T., Lin, Y.-H., jul 2013. An exercise management system for teaching programming. Journal of Software 8 (7), 1718–1725.

[29] Wang, T., Su, X., Wang, Y., Ma, P., feb 2007. Semantic similarity-based grading of student programs. Information and Software Technology 49 (2), 99–107.