Semi-Automatic Assessment of Unrestrained Java Code[*]

A Library, a DSL, and a Workbench to Assess Exams and Exercises

David Insa   Josep Silva
Universitat Politècnica de València
Camino de Vera, s/n
46022 Valencia, Spain
{dinsa,jsilva}@dsic.upv.es

ABSTRACT

Automated marking of multiple-choice exams is of great interest in university courses with a large number of students. For this reason, it has been systematically implanted in almost all universities. Automatic assessment of source code is however less extended. There are several reasons for that. One reason is that almost all existing systems are based on output comparison with a gold standard. If the output is the expected, the code is correct. Otherwise, it is reported as wrong, even if there is only one typo in the code. Moreover, why it is wrong remains a mystery. In general, assessment tools treat the code as a black box, and they only assess the externally observable behavior. In this work we introduce a new code assessment method that also verifies properties of the code, thus allowing to mark the code even if it is only partially correct. We also report about the use of this system in a real university context, showing that the system automatically assesses around 50% of the work.

1. INTRODUCTION

Assessment is an integral part of instruction, as it determines whether or not the goals of education are being met. When assessment works best, it provides diagnostic feedback both to students and teachers. Therefore, it provides both means to guide student learning and essential information for both the learner and the teacher about the learning process.

In the psychology education area, it has been proved that most students often direct their efforts based on what is assessed and how it affects the final course mark (see, e.g., [4], [Chapter 9]). As a consequence, continuous assessment during a course can be used to direct and enhance the learning process. However, providing quality assessment manually for even a small class requires an important effort. When the class size grows, the amount of assessed work has to be limited or rationalized in some other way.

This is the reason why many efforts have been made to produce tools and techniques for the *automatic assessment* (AA). In fact, this has been a hot topic for a long time [7], and there are in the literature many different ways for the teacher to define tests, resubmission policies, security issues, and so forth.

AA has been traditionally focussed on multiple-choice exams. Most of the work has been done in that area, mainly improving the scanning process, allowing recognition of students annotations (e.g., permitting to alter the annotated 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.

Another area of special interest is the correction of programs written by students. Most of the work has focussed on restricted programs, where the student has to select different (finite) options in a GUI to construct a program, or it is based on output comparison. Most advanced AA systems compile the student code, execute it, and compare the result with the correct answer. Random test generation can be also used reducing the number of false positives.

Unfortunately, checking the output of the student's code, even if we compare all possible outputs, is often not enough to ensure a quality assessment. This is illustrated in Example 1.1.

EXAMPLE 1.1. *An exercise says: Implement a class "Car". Cars have an attribute* `numberPlate` *that can be obtained with method* `getNumberPlate()`. *We can assume that this exercise is part of a wider model to represent a garage.*

```
    SOLUTION 1               SOLUTION 2

class Car {              class Car extends Vehicle {
  int numberPlate;
  Car(int np){             Car(int np){
    numberPlate = np;        super.numberPlate = np;
  }                        }
  int getNumberPlate(){    int getNumberPlate(){
    return numberPlate;      return super.numberPlate;
  }                        }
}                        }
```

[*]This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII2015/013. David Insa was partially supported by the Spanish Ministerio de Educación under FPU grant AP2010-4415.

While both solutions could be acceptable for a first-year student, the first solution is unacceptable for a second-year student. Solution 2 is more maintainable and reuses code via inheritance. Solution 1 does not make use of inheritance, and thus it can be wrong when an object "Car" should behave as a "Vehicle". However, both cars have the same attributes and methods, and thus for most tests, they will always produce the same output. In this example, a teacher would not be interested in the output, but in one specific property of class "Car" (i.e., it extends class "Vehicle").

Another important restriction of output comparison is that they just return the whole code as correct or the whole code as wrong. No intermediate mark is possible. The reality is however different: Teachers normally divide an exam into small appraisable pieces. For instance, using inheritance could be assigned a value of 3 points over 10, thus, `Solution 1` should be marked with 7 points.

Moreover, output comparison is very sensible to small errors. For instance, if the student accidentally changes **`return numberPlate`** by **`return numberPate`**, then all output comparison based methods will mark the exam with 0 points. Nevertheless, many teachers would see the typo as something not important or just due to lack of time, and they would mark the exam with, e.g., 9 points. Of course, output comparison methods cannot handle programs that do not compile (e.g., because a semicolon is missing).

Our approach goes beyond output comparison, and it can assess unrestrained code (i.e., the student can freely use the complete programming language without restrictions). In particular, our technique can automatically assess the code not only from the final output, but also checking whether the own source code fulfills any properties desired by the teacher (e.g., definition of a particular class hierarchy, implementation of interfaces, existence of a particular field, etc.). Concretely, we have developed an assessment library that uses the abstraction and advanced meta-programming features of Java to allow properties verification. Based on this library, we have developed a domain specific language (DSL) for the specification of automatic exam assessment templates. And we have also developed a system that integrates the DSL into a workbench with GUIs and facilities for AA of unrestricted Java programs.

We believe that our approach opens a new line of research in AA, and we have made the library, DSL, tools and the source code open and free. In contrast to many other approaches [7], it has not been designed as a tool for a specific Java programming course useful for a single teacher, but a tool that can be widely used, configured and augmented to design any Java exam or exercise and to automatically assess batteries of exams.

It is important to note that we call our tool *semi-automatic* even though it is as automatic as the other tools and systems available in the bibliography that claim that they are automatic. We want to remark that not all programming exercises can be automatically assessed. In the general case, AA is an undecidable problem, and thus no fully-AA-system can be constructed. For instance, a program that does not terminate cannot produce an output. And it must be stopped. For this program, it is not possible to know whether it is really non-terminating or just inefficient, because determining non-termination is by itself an undecidable problem. Other programs simply do not compile, they are syntactically wrong, and thus no dynamic analysis can be done with them. For all these kinds of problems, human (teacher) intervention is needed to correct the program or just to assign a mark.

Our system also performs output comparison. It automatically compares the output of the exam with the expected solution. It also uses test generation to compare the output with many different inputs. But, as a new feature, it also validates arbitrary properties of the code. Whenever human intervention is needed, the system prompts the teacher with the available information showing the problem found, and both the student's exam and the solution.

We summarize the main contributions of our work:

- An assessment Java library with abstraction methods for the verification of properties in Java code.
- A DSL built on top of the library for the specification of exams and their corresponding assessment templates.
- A semi-automatic assessment tool for Java exams and exercises based on output comparison via tests generation and verification of properties.
- A report on teachers assessment problems identified with our tool.

2. RELATED WORK

Pears et al. [12] classify the tools that support teaching programming into four groups: (1) visualization tools, (2) AA tools, (3) programming support tools, and (4) microworlds.

Focussing on AA tools, the main area of interest has been automatic marking of multiple-choice exams. This is of special interest in university courses with a large number of students, and the problem has been solved and solutions implanted in almost all universities. Some remarkable systems of this kind are [14, 6].

A second kind of AA tools is based on *visual algorithm simulation exercises* [9]. These tools use advanced GUIs that control all possible (finite) actions of the student 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) [9].

In this paper we focus on a third kind of AA tool that has been less investigated: AA of source code. In this area, almost all approaches are based on output comparison (see, e.g., [13, 5], and the four surveys [2, 1, 10, 7]). The general idea is to compile the students code, execute it with a predetermined set of inputs, and compare the outputs with the expected results. The most advanced systems [15, 8, 3] 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 use the compiler as a parameter, and they just compare the results.

Unfortunately, there are very few works that perform code analysis to mark exams. Two exceptions are [11] and [16].

In both approaches, the idea is to measure the similarity of the student code to the pool of known solutions. Similarity is computed with a graph representation of the code.

A comparison of the most important AA tools can be found in [7]. Other previous reviews are [2, 10, 1].

Our approach also uses output comparison with automatic test generation. But, in contrast to many of the related work, we do not use a pool of known solutions, a model, or a graph. We use a solution to the exam provided by the teacher. We compile it and use it as an oracle to generate test cases. Moreover, our generated tests are not completely random, they are based on an analysis of path conditions to maximize code coverage.

Another important difference of our approach is the property verification module. We are not aware of any other technique that uses a DSL to specify properties that can be automatically validated and assigned a mark. Finally, another feature that is completely novel, is the possibility to mark an exam that does not compile.

3. THE SYSTEM IN A NUTSHELL

Our automatic assessment system, including libraries, the DSL specification, the workbench, manuals, the source code, and examples, is free and publicly available at:

`http://www.dsic.upv.es/~jsilva/teaching/ASys/`

We describe here its functionality and architecture, and we refer the interested reader to the project website for implementation details. Essentially, our system performs three different assessment phases:

1. *Compilation*: To identify compilation errors.
2. *Analysis*: To check whether the specified requirements are properly implemented.
3. *Testing*: To identify runtime errors.

Phases 1 and 3 are already implemented and reported in other works (see, e.g., [15, 8, 3]) so we will explain here phase 2, which is a novel contribution of our work. We just want to remark that, for phases 1 and 3, our system is not limited to a set of predefined tests to check some provided finite input-output pairs. Contrarily, our system can generate any arbitrary number of test cases. For this, the system inputs the solution of the exam (in Java), and iteratively generates them. Then, tests are used with a battery of Java exams, so that, the output of exams is checked against the output of the compiled solution. Tests implemented manually can also be used in the testing phase.

Phase 2 is our most important contribution. To implement the analysis phase, we first implemented a Java metaprogramming library with assessment functions, and a DSL on top of the library for the specification of assessment templates. Because the DSL is compliant with Java, it is directly executable by an assessment engine that can automatically verify the properties in the template against a given exam. In the next sections we describe the library, the DSL, and the workbench that integrates all of the components.

3.1 The assessment library

One of the most important contributions of our work is our assessment library, because it is per se a tool to create assessment systems. It is a Java library with heavy use of reflection that provides an API composed of 77 methods that can be used to inspect and analyze Java source code. The library provides a class called "Inspector" that internally makes all the abstraction. This class provides several methods to query and manipulate the meta information in order to check properties of the code.

EXAMPLE 3.1. *Some methods of the library, useful to determine whether a given property is satisfied, follow:*

-
  ```
  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] stands for the generic type S).

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

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

EXAMPLE 3.2. *Given the code in Solution 1 of Example 1.1, with the library, we can automatically check whether class "Car" extends class "Vehicle" with the following code:*

```
public boolean checkCarExtendsVehicle() {
  Class<?> carClass = inspector.getClass("Car");
  Class<?> vehicleClass = inspector.getClass("Vehicle");
  boolean carExtendsVehicle =
       inspector.checkSuperClass(carClass, vehicleClass);
  return carExtendsVehicle;
}
```

where methods `inspector.checkSuperClass` *and* `inspector.getClass` *are provided by the library and have the obvious meaning.*

The assessment library is used by our system, but it is an independent piece of work that can be used in other systems and projects. Therefore, we have made it available independently as an standard Java library. The library is publicly accessible at: `http://www.dsic.upv.es/~jsilva/teaching/ASys/library/`.

3.2 The assessment DSL

With the assessment library, we have created a DSL that allows teachers to programmatically assess Java exams and exercises. The DSL allows us to load a Java program (e.g., an exercise) and check whether it has been implemented correctly. For instance, we can check the correct use of inheritance, interfaces, abstract classes, etc.

In particular, the DSL provides mechanisms to specify properties and assign marks to them. Each exam is assigned with a set of evaluation pieces, and each evaluation piece is

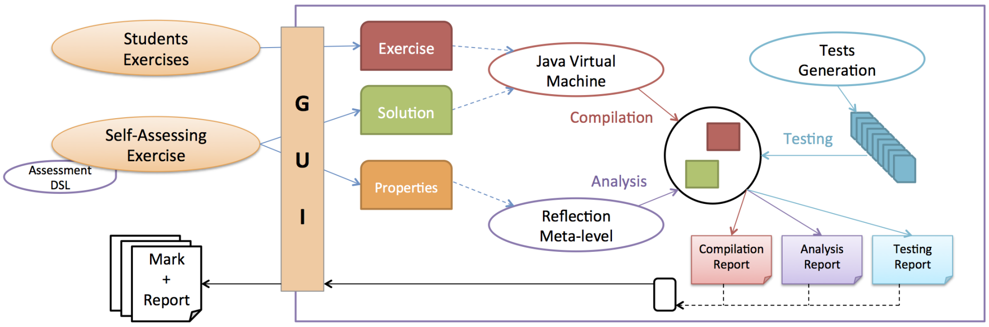


Figure 1: Architecture of ASys

associated with one property that we want to check, and a mark that is assigned if the property holds. Properties can be specified using the library API.

When properties hold, the correction can be done automatically, assigning the specified mark to each property. When a property fails, the system can be configured to either assign a mark (e.g., 0 points), or work semi-automatically: the teacher is prompted with the source code that caused the problem, the explanation of the property that fails, and a form to assign a mark and a review for this failing property. The DSL also gives support to this behavior.

Concretely, all functionality of the DSL (creating assessment templates formed from evaluation pieces) has been accompanied with a GUI, so that teachers can graphically create assessment templates and include properties and marks using buttons and text boxes.

Due to lack of space, we refer the reader to the project website for explanations and examples of the DSL.

3.3 The assessment workbench

Our semi-automatic assessment system is called ASys (which stands for Assessment System). The architecture of ASys is depicted in Figure 1. The input of the system is (i) a path to the directory with the exams to be assessed, (ii) a path to the directory with the solution to the exam, and (iii) a marking template specified with the assessment DSL that defines how to mark the exam.

The output of the system is, for each exam, a report that specifies the final mark together with a detailed list of the problems found. This output is useful for both the teacher and the student, and thus, each report is duplicated and presented in two different ways: one for the teacher with information useful to automate the marking and publication of marks, and another for the student, with learning feedback explaining the problems encountered. Concretely, the report is composed of information from the system and the teacher about compilation errors (compilation), unsatisfied properties (analysis), and runtime errors (testing).

In order to understand how this triple report is built, we can see the data flow diagram (DFD) of the system depicted in Figure 2. Dark boxes with `Exam`, `Exam solution`, and `Assessment template` are the inputs provided by the user. They correspond to those in Figure 1. Following the paths in the DFD, one can easily understand that there are three different phases highlighted with the `Phase 1`, `Phase 2`, and `Phase 3` boxes. These phases correspond to the three modules in the architecture: `Compilation`, `Analysis`, and `Testing`.

One could think that the three phases are executed sequentially. That is, the exam is compiled once, then, the generated code is analyzed, and, finally, it is tested to find runtime errors. This sequential behavior is the one used in previous tools (without the analysis phase, which is novel). However, our system works differently.

We repeat the three phases every time that the teacher modifies the code; and this happens every time that an error (either compilation error, unsatisfied property, or failing test) is identified. Hence, the general schema in the DFD is: `find one error` → `prompt the teacher` → `re-compile the whole exam` → `check *all* properties` → `check *all* tests`. This is repeated until no more errors are found. This schema:

- allows the teacher to correct (and mark) any part of the code when an error is found. He/she can correct a single error, more than one error, or even the complete exam at once. Then, Asys will check again that everything is correct.
- prevents the teacher to introduce errors. For instance, once a property A has been already corrected by the teacher, the correction of property B could make A to fail again. But this is not a problem, because Asys recompiles and verifies all properties again.

Figure **??** shows a screenshot of ASys when reporting an error: "*class QuadrangularPrism should extend class Square*". In the left window, we can see (in different tabs) the source code of the student (automatically showing the cause of the error). This code must be modified by the teacher to solve the problem. For this, at the right window, we can see the original source code of the student, and the solution. Once corrected, at the bottom, the teacher can assign marks to the problems encountered, and reviews, which will be included in the final report for the student. Reviews are (optionally) used to provide feedback explaining the cause of the error. Observe that several (independent) marks and reviews can be added to the same property (e.g., because several problems are encountered). Note also that "review" is a listbox. It contains all previous reviews for this property. This is very useful when correcting several exams, because once a problem is identified and a review introduced, it can be reused in different exams. This significantly speeds up the assessment, because after a few exams, the errors are often repeated once and again; and the teacher only has to clic and select to automatically assign a review and a mark.

4. A USE CASE

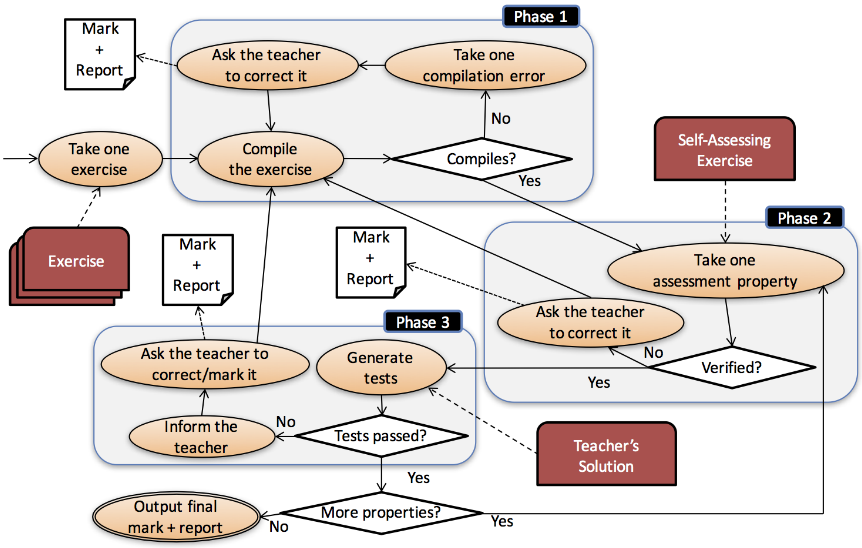


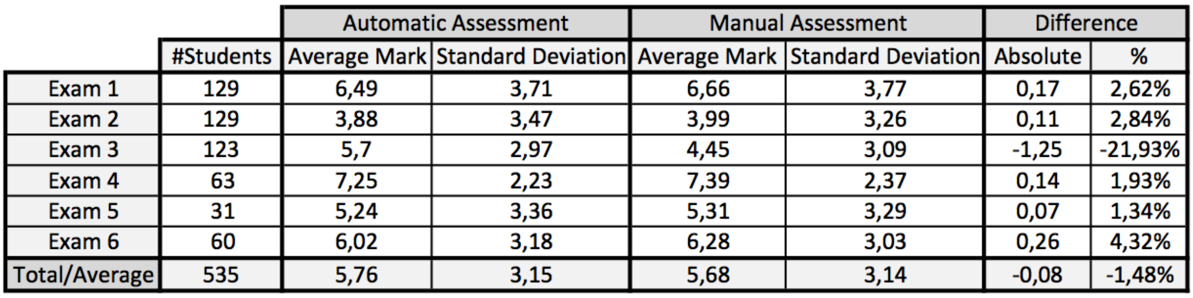
Figure 2: Data Flow Diagram of ASys

The use case reported in this section is not an artificial one. It is a real example performed by different teachers (different from the authors) in a real university exam. Three Java exams were done during a second year university course at Universitat Politècnica de València. There were 5 teachers and 381 students involved. As normally, each of the teachers assessed a subset of the exams, and they published the marks. Then, we asked them to assess again the exams with ASys. Results are summarized in Figure 3.

In the table, each row is a different exam. The meaning of the columns is the following: Column `#Exams` contains the number of exams to be assessed. Column `#Properties` contains the number of properties in the assessment template. Column `Automatic` represents the percentage of properties automatically corrected by Asys (without teacher intervention). Column `Average Mark` contains the average mark of all exams (this is computed for the teachers assessing alone and assessing with Asys). The standard deviation of the marks is computed in column `Standard Deviation`. Finally, column `Difference` computes the difference in absolute value and percentage between the two average marks computed.

The first important conclusion is that ASys is able to automatically perform almost half of the assessment work (48%). Moreover, it is important to note that the average mark obtained with ASys and without ASys is different. The difference is small in two exams (less than 3%), but it is significant in the third exam (more than 21%). We studied with the teachers the causes of these differences. In almost all cases they were due to errors of the teachers in the first (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 in the first teacher assessment were due to (1) wrong code introduced by the students in classes not involved in the exercise (and thus, no revised by the teacher and no penalyzed), (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) correct code very difficult to understand even for the teacher, (6) introduction of dead code, or even (7) errors of the teacher when marking (this happened massively in the third exam and produced the difference of 21.93%: One teacher wrongly marked one question for all students).



		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	0,17	2,62%
Exam 2	129	3,88	3,47	3,99	3,26	0,11	2,84%
Exam 3	123	5,7	2,97	4,45	3,09	-1,25	-21,93%
Exam 4	63	7,25	2,23	7,39	2,37	0,14	1,93%
Exam 5	31	5,24	3,36	5,31	3,29	0,07	1,34%
Exam 6	60	6,02	3,18	6,28	3,03	0,26	4,32%
Total/Average	535	5,76	3,15	5,68	3,14	-0,08	-1,48%

Figure 3: Performance results obtained after assessing real exams with ASys

In the same way that we, teachers, make mistakes when programming, we also make them when correcting a program. Tool assistance can help not only to make part of the work automatically, but also to improve the quality of our assessment. In this way, ASys allowed to identify many assessment errors, and, at the same time, it introduced a systematic (partially automatic) methodology to assess the exams, providing also reports for the student and the teacher.

5. CONCLUSIONS

We have presented a new tool for semi-automatic assessment of Java code. This tool introduces a new analysis phase that can verify properties and assign a mark to them. The tool is built using a DSL defined over a new meta-programming library defined to assess Java code. The new system has been used in the university with very good results. It was able to assess more that 380 exams with several different problems including compilation errors.

As already reported in [7], it is surprising, and quite disappointing, to see how few AA systems are open-source, or even otherwise (freely) available. In many papers, it is stated that a prototype was developed but we were not able to find the tool. In some cases, a system might be mentioned to be open source but you need to contact the authors to get it. This is the cause why we are reinventing the wheel, implementing the same assessment systems once and again. For this reason, we made our system, DSL, and library open-source and publicly available, so that other researchers can reuse it or join efforts to further developing it.

6. REFERENCES

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

[2] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. In *Computer Science Education*, volume 15, pages 83–102, 2005.

[3] C. Beierle, M. Kula, and M. Widera. Automatic analysis of programming assignments. In *Proc. der 1. E-Learning Fachtagung Informatik (DeLFI ',03)*, volume P-37, pages 144–153, 2003.

[4] J. Biggs and C. Tang. Teaching for Quality Learning at University : What the Student Does (3rd Edition). In *Open University Press*, 2007.

[5] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. CodeWrite: Supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 09–12, 2011.

[6] R. Hendriks. Automatic exam correction. 2012.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppala. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.

[8] H. Kitaya and U. Inoue. An online automated scoring system for Java programming assignments. In *International Journal of Information and Education Technology*, volume 6, pages 275–279, 2014.

[9] M.-J. Laakso, T. Salakoski, A. Korhonen, and L. Malmi. 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*, pages 28–36, 2004.

[10] Y. Liang, Q. Liu, J. Xu, and D. Wang. The recent development of automated programming assessment. In *Computational Intelligence and Software Engineering*, pages 1–5, 2009.

[11] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking student programs using graph similarity. In *Computers & Education*, volume 54, pages 545–561, 2010.

[12] A. Pears, S. Seidman, C. Eney, P. Kinnunen, and L. Malmi. Constructing a core literature for computing education research. In *SIGCSE Bulletin*, volume 37, pages 152–161, 2005.

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

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

[15] S. Tung, T. Lin, and Y. Lin. An exercise management system for teaching programming. In *Journal of Software*, 2013.

[16] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. In *Information and Software Technology*, volume 49, pages 99–107, 2007.