Document downloaded from:

http://hdl.handle.net/10251/176318

This paper must be cited as:

Insa Cabrera, D.; Pérez-Rubio, S.; Silva, J.; Tamarit Muñoz, S. (2021). Semiautomatic generation and assessment of Java exercises in engineering education. Computer Applications in Engineering Education. 29(5):1034-1050. https://doi.org/10.1002/cae.22356

The final publication is available at

https://doi.org/10.1002/cae.22356

Copyright John Wiley & Sons

Additional Information

Semi-Automatic Generation and Assessment of Java Exercises in Engineering Education

David Insa, Sergio Pérez*, Josep Silva, and Salvador Tamarit

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Valencia, Spain

{dinsa,serperu,jsilva,stamarit}@dsic.upv.es

Abstract

Continuous assessment is essential in education. It should be an integral part of education that provides immediate feedback to students. Unfortunately, the assessment of programming source code is still a hand-operated and error-prone task, which can take weeks before the student gets feedback. This work presents a semi-automatic code assessment method able to automatically apply black-box assessment (which relies on the comparison of input-output pairs) and white-box assessment (which relies on checking different source code properties). The method proposed is a general-purpose assessment system that was originally designated to be used in engineering education, but that can be used in other educational contexts to assist the assessment of any Java programming assignments or exams. The main advantage of this system is that the assessment made is quicker, exhaustive, and objective; and it does not produce false positives. After the application of this method along two years in several real university courses, we have released a public and free implementation. An empirical evaluation of this system estimates that the amount of assessment work automatically done by the tool is over 48%. Additionally, the system has been used to measure the average subjective influence (i.e., assessment errors) introduced by teachers when they assess exams manually.

Keywords: Automatic assessment, Java, assessment tools, marking, evaluation

1. INTRODUCTION

Programming languages are one of the common subjects in many university degrees. Nowadays, most of the engineering degrees contain at least one subject focused on programming. When addressing the teaching of a programming course for first year students, the teacher usually faces students with different backgrounds (including goals, abilities, qualities, preferences, or previous knowledge). All these factors affect the student's motivation and must be borne in mind when creating the student's materials (Kelly and Tangney, 2006).

Learning programming is a complex process. Some learning models include a self-assessment process measuring the improvement of the students' programming skills (Cedazo et al., 2015; Dogan and Dikbiyik, 2016) while others allow the student to customize the learning process by providing their previous knowledge to the system (Jovanovic and Jovanovic, 2015; Gross and Pickwart, 2015; Gavrilović et al., 2018). There are many different learning systems of diverse natures that provide support to the learning process in different ways. For instance, the study performed in (Chakraverty and Chakraborty, 2020) shows that suitably designed tools based on some learning strategies like visual programming, pair and collaborative programming, or game-based learning among others, increase the motivation of the students. In fact, their study proved that these tools improve the computational thinking and academic performance of the students as well as help the instructors monitoring the students' progress. These systems, where the student is capable of guiding its own learning process, are particularly useful to impart MOOCs (Massive Open Online Courses). MOOCs are online teaching courses that hosts hundreds or even thousands of participants. They have proven to be a valuable source of knowledge by both teachers and students (Sra and Chakraborty, 2018). Due to the large number of participants and the expected short-time feedback, MOOCs of computer programming cannot rely on manual assessment. Manual assessment of source code is an error-prone and time-consuming task (Caiza and Álamo Ramiro, 2013). In fact, despite having exhaustive assessment criteria and test cases available to correct the code, identifying the errors can be a difficult task, particularly for large projects. For this reason, computer programming MOOCs usually provide some kind of automatic evaluation system. To this purpose, MOOC systems use automatic graders to automatically assess all the programs (usually thousands of them) submitted by the students (Jamil, 2017; Marin et al., 2017; Muuli et al., 2017; Bey et al., 2018; Galan et al., 2019; Rahaman et al., 2019; Gordillo, 2019). The automatic grader system is a key element of any MOOC framework. It needs to be able to efficiently assess thousands of solutions fast and accurately. Thus, some factors must be considered when designing a

grader for computer programming MOOCs. This grader must fulfil a set of features like efficiency, security, provide accurate feedback, and many others that are further detailed in (Király et al., 2017; Xiong and Suen, 2018).

Since MOOCs have become a popular mode of learning in the last ten years, the study of automatic assessment (AA) in educational contexts is a research field that has been deeply studied lately (Ihantola et al., 2010; Caiza and Álamo Ramiro, 2013; Souza et al., 2016; Annamaa et al., 2017; DeNero et al., 2017; Pieterse and Liebenberg, 2017; Restrepo-Calle et al., 2019; Buyrukoglu et al., 2019; Bryan, 2019; Shao et al., 2019). The development of an AA system is often based on different key factors (Howatt, 1994; Kiraly et al., 2017). Among these factors, we find the definition of complete test suites to check the output values as a common practice. Many AA tools are based on this criterion (Demir et al., 2010; Souza et al., 2016; DeNero et al., 2017; Liu et al., 2019; Daradoumis et al., 2019; Galan et al., 2019; Gordillo, 2019).

Nevertheless, although being a good indicator, there are many other features of the source code that cannot be verified when simply running the program, like the use of good programming habits such as code style (Naik and Tripathy, 2008), or the reusability and maintainability of the code (Subedhaa and Sridharb, 2012). In AA systems based on output comparison, whenever the solution's output does not match with the expected one, even if the difference is produced by a typo in the code, the solution is considered as wrong and marked accordingly. Moreover, output comparison does not provide information about why the code is wrong. Regretfully, even if all possible outputs of a student's code are checked, a quality assessment is not guaranteed. Example 1 illustrates this statement.

Example 1. Consider the following Java exercise:

> The class diagram of Figure 1 models geometric figures. Implement a class "*Cube*" inside the model. A Cube contains three decimal fields (x, y, and z) that define its position in the Cartesian plane, and another decimal field (side) whose value can be obtained by invoking method *getSide*(). It also has a method *getVolume*() that returns the volume of the cube. The implemented code must use the studied mechanisms to maximize the reuse and maintainability of the code.

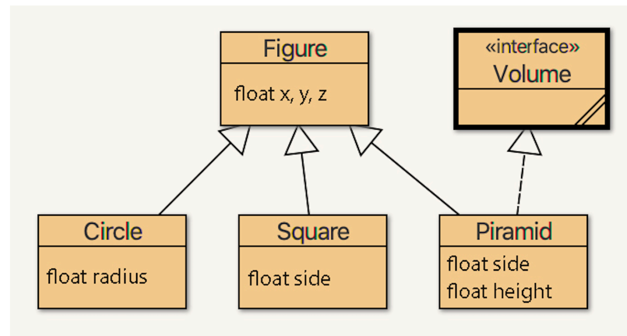
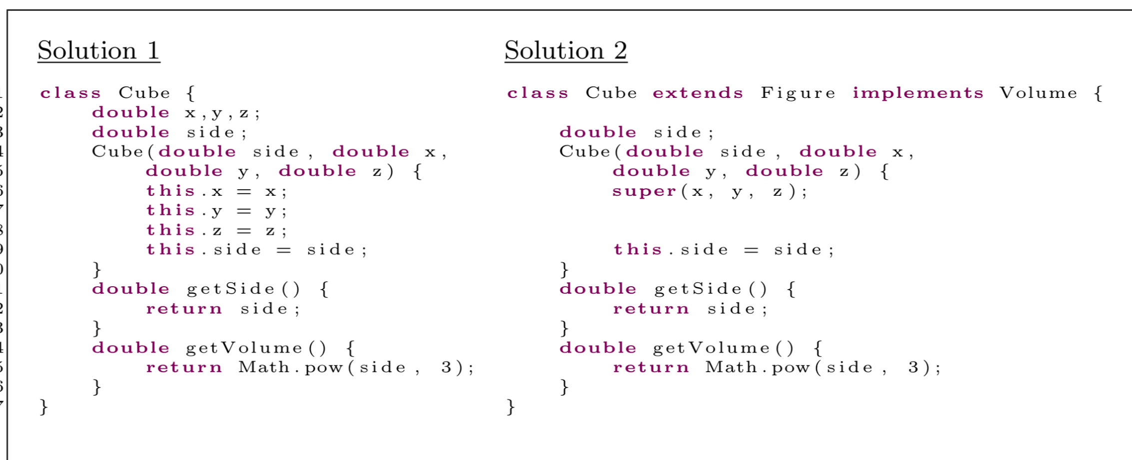


Fig 1: Class diagram of the geometric figures model

This exercise can be considered as a part of a larger model designed to define a collection of geometric figures. Figure 2 shows two possible implementations to solve the exercise.



Solution 1

```
1  class Cube {
2      double x,y,z;
3      double side;
4      Cube(double side, double x,
5          double y, double z) {
6          this.x = x;
7          this.y = y;
8          this.z = z;
9          this.side = side;
10     }
11     double getSide() {
12         return side;
13     }
14     double getVolume() {
15         return Math.pow(side, 3);
16     }
17 }
```

Solution 2

```
class Cube extends Figure implements Volume {

    double side;
    Cube(double side, double x,
        double y, double z) {
        super(x, y, z);


        this.side = side;
    }
    double getSide() {
        return side;
    }
    double getVolume() {
        return Math.pow(side, 3);
    }
}
```

Fig. 2: Different implementations of class `Cube`

While both solutions could be valid for a student in the first course of, e.g., an engineering degree, Solution 1 would not be correct for a student in superior courses. On the one hand, Solution 1 does not apply inheritance, so if the *Cube* object is expected to behave as a *Figure*, the provided solution would be wrong. Additionally, it implements the method *getVolume*() without the use of an interface implementation, then, the system cannot infer if Cube objects implements (or not) a method to calculate its volume. On the other hand, Solution 2 reuses existent code (by means of inheritance), making the code more maintainable. Furthermore, it implements an interface (*Volume*), signifying the existence of a method *getVolume*() for all *Cube* objects. However, both cubes share the same attributes and methods and, thus, they would return the same output for

most test cases. In the example, the teacher might be interested in some properties of class *Cube*, instead of being interested in the computed output (e.g., *Cube* extends class *Figure*, it makes use of inheritance through the use of `super()`, and it implements interface *Volume*).

Output comparison techniques also present another important limitation: because they cannot access the source code and they only check whether the output is correct or wrong, then, they have only two possible evaluation values: correct or wrong. Nevertheless, the real scenario is often different: teachers usually divide the mark of an exercise into smaller evaluable parts (Ahoniemi and Reinikainen, 2006; Gavrilović et al., 2018; Buyrukoglu et al., 2019; Delgado and Medina, 2020). Assuming a methodology where 10 points is the maximum grade, 5 the minimum grade to pass the exam, and each appraisable piece not fulfilled decreases the student's final mark, Solution 1 would barely pass the exam. For instance, not using inheritance and interfaces could decrease a student's mark in 3 and 1 points, respectively, which would give Solution 1 a final grade of 6 out of 10.

In addition, the comparison of outputs is very sensible to small errors. For example, if a student writes `Math.pow(side, 2)` instead of `Math.pow(side, 3)` at Line 15 of Figure 2, then all the methods that employ output comparison will give a mark of 0 to the solution. However, this error may be also seen as a lack of time or due to a distraction for many teachers, who would mark the solution with, e.g., 9 out of 10 points.

1.1 Contributions

We are in line with the idea of (Naik and Tripathy, 2008) that an AA system must evaluate solutions from both dynamic and static aspects, and having the ability to inspect, change, and execute the students' code, exactly as teachers do. We propose a new assessment system of this kind, so that it helps in assessing Java exercises/exams in an exhaustive, assisted, and fair way. The system uses the JavAssess assessment library (Insa et al., 2015) that provides reflection capabilities together with an API to introspect, modify, and test a Java program. To hide the complexities inherent to the use of this advanced library, we have developed a graphical domain specific language (DSL) that uses the library to generate self-assessment exercises.

Thanks to the reflection capabilities, our system goes beyond output comparison and it can not only assess the final output produced by the code, but it can also inspect the structure of the source code itself and

the language resources used in the performed implementation (e.g., the existence of a particular field, the definition of a concrete class hierarchy, the implementation of a particular set of interfaces, etc.). This allows the technique to check any property of the code, producing an intermediate mark when the code is partially correct.

Moreover, the system is designed in such a way that all automatable tasks such as opening the students code, compiling it, detecting the errors, executing test cases and comparing the results with a teacher's solution, storing the marks, etc. have been completely automatised, potentially saving a lot of time. In particular, a correct student's solution is always marked automatically. A student's solution with errors, can be also automatically marked because the system can correct the errors with reflection (intercession), which allows the system to automatically modify the students code.

The main functionalities of the system include the following:

1. Automatic detection of the assessment order. The system uses Algorithmic Debugging (Caballero et al., 2017) to detect the order of assessment.
2. Collaborative assessment. Exercises are divided into atomic properties. When the system detects an error in a property, it records the error, the mark assigned to this error, and the comment introduced by the teacher for the student. When the same error is detected, the system tells the teacher that this error was penalized with mark M. This avoids penalizing the same error in a different way. This is also applied between different teachers, so that the assessment knowledge and work is shared.
3. Individual learning record and suggestions. Student's errors are recorded and classified (e.g., in Java, as polymorphism, abstract classes, interfaces, etc.). Hence, the system can recommend exercises specific to solve the weak points of the student.
4. Personalized output. Besides the solution of the teacher, our system also outputs the student's code corrected.

And it also proposes:

5. Several ideas that must be considered when implementing future AA tools:
    - The inter-dependence of properties must be specified or inferred.
    - AA systems must be able to correct the exercises.

6. A sophisticated and secure choreography to run the compilation and assessment (including testing) processes in the client. This is essential to increase security (the code of the student is executed in its own machine) and also scalability (a big load is distributed among the clients).

Also, one of our main contributions is showing that a system needs to use meta-programming to check structural properties (Király et al., 2017).

7. We have facilitated this by not only implementing it in our system, but also implementing a DSL on top of JavAssess (Insa et al., 2015) that can be used in other systems. For instance, the DSL uses graphical boxes to check, e.g., whether class *Lion* extends *Animal*, and it would generate the following reflective code:

```
String[] projectPaths = { "/path/project/" };
Introspector introspector = new Introspector(projectPaths);
Class<?> lionClass = introspector.getClass("Lion");
Class<?> animalClass = introspector.getClass("Animal");
return checkSuperClass(lionClass,animalClass);
```

Note that all the metaprogramming is already implemented by the *JavAssess* class *Introspector.*

Finally, other contribution of our work is the report of our experiments. This has determined when is beneficial to use AA systems (with more than 17 students), and the linear trends of the manual and automatic assessment efforts (see Figure 11).

The rest of this paper presents these contributions, including a new model and architecture for semi-automatic assessment tools (Section 3.1), a DSL for the definition of self-assessment exercises (Section 3.2.2), an implementation of the proposed model for Java (i.e., a semi-automatic assessment tool for Java exercises and exams) (Sections 3.2.3 and 3.2.4), and a real use case where the tool was used for two academic years and that reports the feedback given by both teachers and students (Section 4). The DSL and tools described in this paper are available at: http://personales.upv.es/josilga/ASys.

2. BACKGROUND

Assessment is essential in education because it allows us to determine to what extent are the educational goals being met and, thus, providing diagnostic feedback to teachers and students. For this reason,

assessment is not only a way to supervise the students' learning, but also a source of information for both student and teacher about the learning progress itself (Ihantola et al., 2010). The assessment of programming exercises has always been considered a time-consuming task for teachers due to the time required to test and correct the code of all students' solutions. Usually, teachers invest a lot of time correcting errors that are repeated over and over by different students, providing them the same feedback independently.

For this reason, the use of AA systems has been studied in last years as a useful tool to reduce teachers' workload. These systems consider different aspects of the code that can be classified as *dynamic*, related to the execution of the code (e.g., the comparison of the output values obtained for both the teacher's and student's solution with a certain set of inputs); or *static*, related to the structure of the code (e.g., the correct use of reusability resources or the coding style) (Naik and Tripathy, 2008). Traditionally, dynamic aspects are associated with automated assessment while static aspects are considered to be assessed manually (Buyrukoglu et al. 2019).

Many tools and systems available in the bibliography describe themselves as *automatic* (Liu et al., 2019; Daradoumis et al., 2019; Galan et al., 2019; Gordillo, 2019; Shao et al., 2019). However, the AA problem is undecidable in general and, hence, a full-AA-system cannot be built. For example, a non-terminating program cannot generate an output (it must be halted). In this case, the reason for the non-termination cannot be determined. It could be due to inefficiency. Determining non-termination is an undecidable problem by itself, so there is no way to automatically assess all programs. Other programs that are syntactically wrong produce compilation errors and, thus, they cannot be dynamically analysed. A human (teacher) mediation is mandatory for these scenarios in order to correct or evaluate the program.

A lot of important AA tools have been analysed and compared in (Pettit et al., 2015; Souza et al., 2016), and there exist also previous reviews that can be found in (Ala-Mutka, 2005, Abd Rahman and Jan Nordin, 2007, Liang et al., 2009; Ihantola et al., 2010; Skalka et al., 2019). Different assessment systems solve some problems related to exercise submissions. Some of them are able to fix syntax errors to compile uncompilable code (Fisher, 1992; Auffarth, 2008), others, apart from providing output comparison with an associated message as feedback, let the teachers to add personalised comments to a particular student (Auffarth, 2008; Benson, 1985; Georgouli and Guerreiro, 2010; Tung et al., 2013), and there are some that analyse the

code structure (Jackson, 2000; Joy et al., 2005) or the code style (Jackson, 2000; Joy et al., 2005; Edwards and Pérez-Quiñones, 2008; Wang et al., 2011).

2.1 Output Comparison

Most systems focus their assessment on the output comparison technique, which compares the output of test cases executed over the teacher and student's code (Yusof et al., 2012). One of them is the widely extended assessment system WebCat (Edwards, 2003) used by more than 20 separate institutions. This system supports multiple languages and provides the option of testing programs developed by other students. Another relevant assessment system is QuizPack (Brusilovsky and Sosnovsky, 2005), with a slightly different assessment process. It generates questions about the trace of a program for a particular execution. In order to do so, the teacher provides a program and selects a variable that wants to be asked for. QuizPack instruments the code, generates an input, and finds out the value of the indicated variable matching the answer given by the student with the correct result. ProgTest (Souza et al., 2011) augments the output comparison with the coverage measurement concept.

2.2 Augmented feedback

Other assessment systems give the student more specific and personalised feedback. This is the case of ALOHA (Ahoniemi and Reinikainen, 2006), which lets the teacher associate rubrics to each property the grader evaluates. Each rubric consists on a mark and a default comment for each correct or incorrect property evaluated by an external grader system. The tool provided by Denny et al. (2014) was also focused on providing more specific feedback to users, but in a different way. This tool was developed to transform compilation error messages to messages expressed in natural language. Interestingly, the study of this research concluded that the expressivity of the message was not relevant for the students in order to understand their own errors.

2.3 Static analysis and dynamic analysis

Most systems in the literature only perform dynamic assessment or they just introduce static assessment for code style or to measure complexity. The most advanced assessment systems are able to infer from the code the use of good programming practices that enables maintainability or reusability. The tool JACK

(Striewe et al., 2009) is one of them. It allows for the definition of specific properties related to Java programs. These properties are specified and verified according to a language based on the abstract syntax tree (AST) of the program, which is unique for a particular Java program. A new similar assessment system is the one introduced by Delgado and Medina (2020) for C/C++ programs. They developed a C library able to inspect the static structure of the code. The teacher just introduces a list of properties in a check file by calling certain functions of the library and the tool is able to check whether these properties hold in the student's code. After testing the student's code dynamically and statically, the system provides specific and personalised feedback to the student describing the non-checked properties. The eGrader assessment system (Shamsi and Elnagar, 2012) also performs static and dynamic analysis over students' submissions. Dynamic analysis is performed by JUnit while static analysis is based on the Control Dependence Graph and Method Call Dependencies, which are constructed from the AST. In their model, structure and software metrics are specified along with control structures' positions and represented as a code called *Identification Pattern*. This system efficiently and accurately grade submission with semantic error and generates detailed feedback for each student providing them a report for the overall performance for each assignment. Algo+ (Bey et al., 2018) is a system that requires a pool of solutions corrected by a teacher to correct and mark new exams. It uses correct solutions to mark exercises and wrong solutions to provide feedback to the user. This system's performance increases with the number of corrected exercises in the pool, so it is quite useful in MOOCs, where there is a large number of exercise submissions. Another recent AA system of this kind was proposed by Buyrukoglu et al. (2019). In their approach, according to a similarity threshold, similar codes are grouped. At assessment time, the teacher only needs to assess and comment one exercise of each group, the rest of the exercises of the same group are automatically assessed and the comments are propagated.

The system we present in this paper mixes different features of the described systems. First, it needs the teacher’s solution and a set of properties defined in an intermediate language as it happens in the JACK system. Unlike some of the described systems, the exercise does not need to be provided with a set of test cases, our system is able to generate its own test suite maximizing branch coverage. Additionally, it uses an assessment DSL that performs dynamic and static analysis of the source code measuring the number of tests passed and properties fulfilled and associating a recommended mark for them, like Delgado and Medina (2020). The DSL also allows the system to automatically correct errors, so they do not affect the rest of the code. Finally, the

system only requires the intervention of the teacher if a property fails. In such a case, the system recommends a mark and a feedback message based on previous markings done not only by the current teacher, but by any teacher.

3. MATERIALS AND METHODS

3.1 A semi-automatic assessment model

As it happens in the compiler construction context, where compilation time and runtime are distinguished, our model also distinguishes between two moments in the assessment process: exercise construction time and assessment time. Figure 3 shows a high-abstraction level scheme with the components of the system.

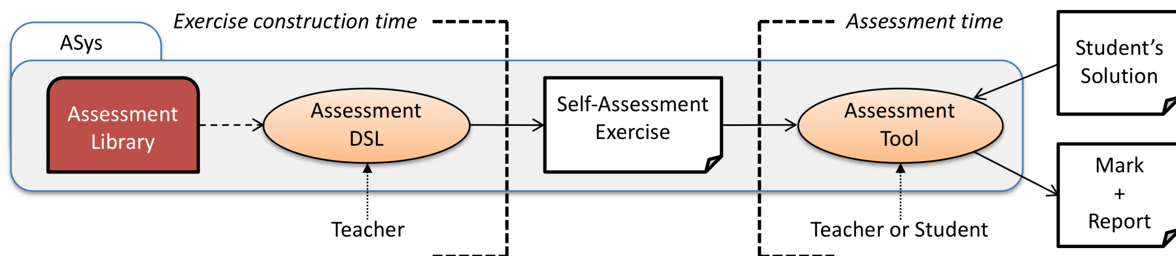


Fig. 3: Semi-automatic assessment scheme

Traditionally, in massive courses (e.g., with hundreds of students), much of the teacher's effort is made at assessment time. For instance, in a university course (this will be explained in detail in Section 4) we measured that the teachers used an average of 1 hour to design an exam and 12 hours to assess it. Our methodology significantly reduces (up to 60%) the effort made at assessment time by increasing the effort made at exercise construction time (up to 100%). This not only produces a quicker assessment process, but it also produces a more reliable and fair process, because a big part of the assessment is made automatically.

In Figure 3, it can be seen that our system is made up of two main independent components:

1. **A DSL for the specification of self-assessment exercises**. It is a graphical language integrated into a GUI that allows teachers to create and generate self-assessment exercises. It internally uses the meta-programming library JavAssess.

2. **A tool that allows for AA of Java programs**. This tool can also be used by students. Therefore, they can self-assess their solution and know their mark straight after an exam; but more interestingly, the tool also allows the student to interactively solve the problems, thus producing a valuable learning resource.

Note that the output of the first phase (*exercise construction*) is a self-assessment exercise, which is the input of the second phase (*exercise assessment*). A self-assessment exercise is the key idea of this scheme: it is really a sophisticated program whose execution automatically evaluates, tests, and compares the student's code against the solution. Therefore, it contains the exercise statement (e.g., a PDF), the source code of the teacher's solution, test cases, and code to orchestrate the assessment.

Our concrete implementation of this model is called ASys, and it is described in the next section.

3.2 ASys: A semi-automatic assessment system

Our assessment system, including manuals, examples, the workbench, the DSL specification, and libraries is publicly available at: http://personales.upv.es/josilga/ASys

In this section we describe the architecture and the functionality of our system. For further implementation details, we refer the reader to the project's website where technical details can be found. In the following subsections, the assessment library, the assessment DSL to generate self-assessment exercises, and the semi-automatic assessment tool are explained in detail.

3.2.1 The Assessment Library JavAssess

Our system is constructed over a metaprogramming library called JavAssess (Insa et al., 2015). The complete API of JavAssess is available at: http://personales.upv.es/josilga/JavAssess/.

3.2.2 The Assessment DSL

Self-assessment exercises can be generated from a specification made with a DSL. The DSL is a compositional graphical language that allows us to specify properties that a Java program must fulfil (e.g., the type of a method) at a high abstraction level (i.e., without the need to explicitly use the API of the library, and

without the need to have knowledge about reflection). The DSL can also associate a mark with a given property. Moreover, properties can be decomposed into other properties, so that marks can be assigned compositionally.

Example 2. Given the code in Solution 1 of Example 1, the following assessment properties could be specified:

1. Implement class *Cube*: 10 points
   1.1. *Cube* uses inheritance correctly: 3 points
      1.1.1. *Cube* extends *Figure*: 1 point
      1.1.2. *Cube* reuses inherited attributes x, y, and z: 2 points
   1.2. *Cube* implements interface *Volume*: 1 point
   1.3. Constructor correctly implemented: 2 points
   1.4. Method *getSide*() correctly implemented: 2 points
   1.5. Method *getVolume*() correctly implemented: 2 points

Figure 4 shows a screenshot of how to specify these properties in ASys using the DSL. On the left, there is a pane with all the specified properties. "Cube extends Figure" is selected, thus, in the bottom-left corner more information about this property is displayed. E.g., the class that should implement this property is `Cube`, and a maximum mark of 1.0 points is assigned to this property. In the central floating pane, a collection of methods is shown. They are the complete API of JavAssess. When a method is selected—in the figure `checkType` is selected—then all variants (overloads) of this method are displayed. When the "Add method" button is clicked, a new line is added to the property specification. In the figure, in the top-right corner, there are three lines where the user only has to fill in the textboxes. They just select class `Cube` and class `Figure`, and check whether the superclass of `Cube` is `Figure` (if line 3 had a `False` instead of a `True` this property would check that `Figure` is not a superclass of `Cube`). These three lines are enough to automatically generate all the associated Java code that searches for the files, analyses the source code, checks the property, and assigns a mark. For instance, the code generated to check the property specified by the three lines in the figure is method `checkCubeExtendsFigure`, shown in Figure 5.

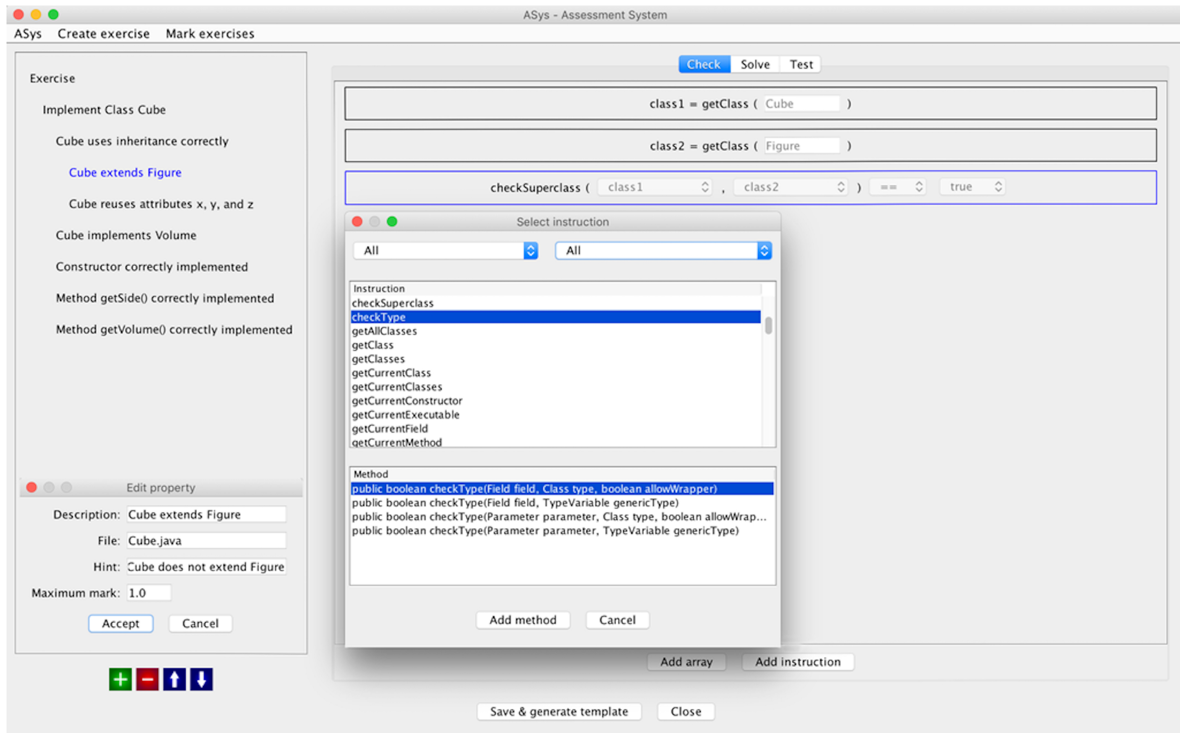


Fig. 4: ASys: specifying properties with the DSL at exercise construction time

When the method returns `False` (either because class `Cube` or class `Figure` does not exist, or because `Figure` is not the superclass of `Cube`) then ASys subtracts the assigned mark to this property (1 point) and tries to correct the property automatically. The code generated to correct this property is shown in method `correctCubeExtendsFigure`.

```
1  import codeassess.javassessment.*
2
3  private String[] projectPaths = { "/path/project/" };
4  private String[] solutionPaths = { "/path/solution/" };
5  private Introspector introspector = new Introspector(projectPaths);
6  private Intercessor intercessor = new Intercessor(projectPaths, solutionPaths);
7
8  public boolean checkCubeExtendsFigure() {
9      Class<?> cubeClass = introspector.getClass("Cube");
10     Class<?> figureClass = introspector.getClass("Figure");
11     boolean cubeExtendsFigure = introspector.checkSuperclass(cubeClass, figureClass);
12     return cubeExtendsFigure;
13 }
```

```
1  public void correctCubeExtendsFigure() {
2      Class<?> cubeClass = introspector.getClass("Cube");
3      Class<?> figureClass = introspector.getClass("Figure");
4      intercessor.setSuperclass(cubeClass, figureClass);
5  }
```

Fig. 5: Code generated by the DSL to automatically assess (top) and correct (bottom) an exercise at assessment time

As shown in Figure 5, besides specifying properties and assigning marks to them, the DSL provides mechanisms to also correct the properties automatically. This can be done thanks to the meta-programming and reflection capabilities of Java, which allows us to modify the source code at runtime.

We affirm that the ability to correct the code is a requirement to achieve higher accuracy in marking, and this is one important difference with other tools. We state that:

```
1  class Cube {
2      double side;
3      double getArea() {
4          return 2 * this.side;
5      }
6      double getVolume() {
7          return this.getArea() * this.side;
8      }
9  }
```

Fig. 6: Implementation of `Cube` with methods `getArea()` and `getVolume()`

Proposition 1. An automatic assessment tool must be able to correct the errors made by a student, not only to provide feedback, but to properly assess the code.

Example 3. Consider the code depicted in Figure 6 where method `getArea()` is wrong. Clearly, all test cases used to check `getArea()` and `getVolume()` will fail. Nevertheless, `getVolume()` is perfectly implemented.

There are two main reasons why the student's code should be corrected rather than merely marked:

- **Making independent the assessment of different properties**. A student's error can propagate its effects to other parts of the code, which will behave incorrectly until the error is corrected. In practice, this means that an error in one method (respectively property) can produce other methods (respectively properties) to fail, thus penalizing one single error more than once if it is not corrected.

  Example 4. Consider again the code problem explained in Example 3. To avoid this problem, ASys would detect that `getVolume()` depends on `getArea()`, thus `getArea()` would be assessed first. Then, ASys would detect that `getArea()` is wrong. ººº11ºº1º Then, the code of `getArea()` would be corrected and, finally, `getVolume()` would be assessed, and evaluated as correct.

- **Especially valuable feedback**. When a student submits a solution with errors, most tools (e.g., Liu et al., 2019; Gordillo, 2019) return a mark and the teachers' solution. But ASys does something much more useful

for the student: it also returns the student's code corrected—note that this is different from returning a solution that contains a code implemented by the teacher—.

As shown in Example 4, ASys can automatically correct properties. But this is often not enough. The order in which properties are corrected is essential to ensure a good assessment. This is captured by the following proposition:

Proposition 2. An automatic assessment tool must be able to detect dependencies between the properties of a system, and stablish an assessment order accordingly to properly assess the code.

Our system uses the principles of a technique called *Algorithmic Debugging* (Caballero et al., 2017) to identify the methods that should be assessed (and corrected) first.

3.2.3 Architecture of the assessment tool

Once a self-assessment exercise has been created with the assessment DSL, an assessment tool is used to load this exercise (see Figure 3). The assessment tool has been implemented both as a web tool, and as a desktop tool. Both behave in the same way, and present an interesting property: the assessment (including compilation and testing) is always made in the client. This is essential to increase security and scalability. The web tool is able to access the JVM through a service with a secure connection with the browser.

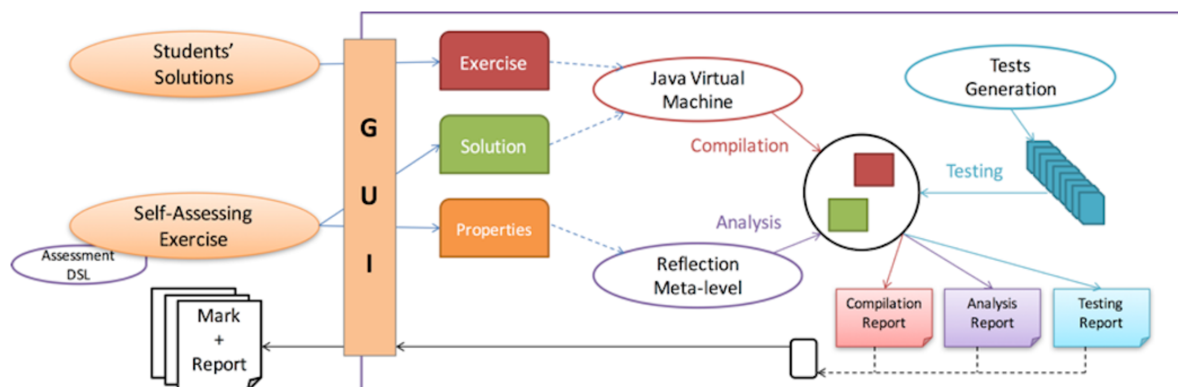


Fig. 7: Architecture of ASys

The internal architecture of the assessment tool is shown in Figure 7. The inputs of the system are (i) a self-assessment exercise created with the assessment DSL, and (ii) the solutions to be assessed. The output of the system is, for each solution, (i) the final mark, (ii) a detailed report of the problems found including the

penalty in the mark for each error, (iii) the teacher's solution, and (iv) the corrected student's source code. This output contains important information for both student and teacher, and for this reason, each report is presented in two different views: one view for the teacher, which shows information that ease the marking process and the publication of marks; and another view for the student, which contains a description of the identified problems, a useful learning feedback for the student.

All this information comes from three sequential assessment phases:

1. *Compilation*: To identify compilation errors.
2. *Analysis*: To check whether the specified properties are fulfilled.
3. *Testing*: To identify runtime errors.

- **Phase 1: Compilation**. The system compiles a student's source code normally, and it shows the user the exact class that produced the error, highlighting the lines and columns of the error.
- **Phase 2: Analysis**. It analyses the source code to verify properties. Because the DSL is compiled to Java, self-assessment exercises are directly executable and can verify the properties specified in the exercises automatically.

  The verification of properties is performed with Algorithm 1. A call to the `correctSolution` function makes the algorithm check all properties against the student's solution. This is made with the auxiliary function `checkProperties` (line 5), which checks whether every single property is satisfied, and whether it passes the tests. If one property fails, then it is automatically corrected with function `solve` (line 16). If function `solve` cannot automatically correct the code (line 17), the wrong code is returned to the main function (line 18), which prompts the teacher the code and asks for correction (function `askTeacherToCorrect`, line 7). When all the properties and tests are satisfied, then `checkProperties` returns a pair `{⊥,code}` (line 22) where code is the final (corrected) code.

Algorithm 1 Correct solution

```
    Input: The student's solution and the template to correct it
    Output: The student's solution corrected
 1: function correctSolution(template, solution)
 2:     properties = getProperties(template)
 3:     code = solution
 4:     repeat
 5:         { property, code } = checkProperties(properties, code)
 6:         if property ≠ ⊥ then
 7:             code = askTeacherToCorrect(property, code)
 8:         end if
 9:     until property == ⊥
10:     return code
11: end function
12: function checkProperties(properties, code)
13:     for all property ∈ properties do
14:         if not check(property, code) or not test(property, code) then
15:             previousCode = code
16:             code = solve(property, code)
17:             if not check(property, code) or not test(property, code) then
18:                 return { property, previousCode }
19:             end if
20:         end if
21:     end for
22:     return { ⊥, code }
23: end function
```

– **Phase 3: Testing**. In this phase, a student's code is tested. Manual tests provided by the teacher with a maximum mark assigned to each of them can be automatically checked. However, our system implements a new test generation module to automatically generate any arbitrary number of test cases. These test cases are used to compare the behaviour of the student's and teacher's solutions. Figure 8 summarises the steps performed by our system to generate these additional test cases.

Firstly, (step 1) both the student's and teacher's solutions are compiled. For each desired method to be tested, the teacher's solution is then (step 2) analysed by a static code analyser, which generates valid inputs whose execution finally invokes the desired method. For each input generated in step 2, we do execute (step 3) the teacher's solution and calculate its output. Therefore, every pair input-output is used to generate a test case. Then, a test case validator (step 4) checks whether all test cases are passed by students' solutions, producing reports.

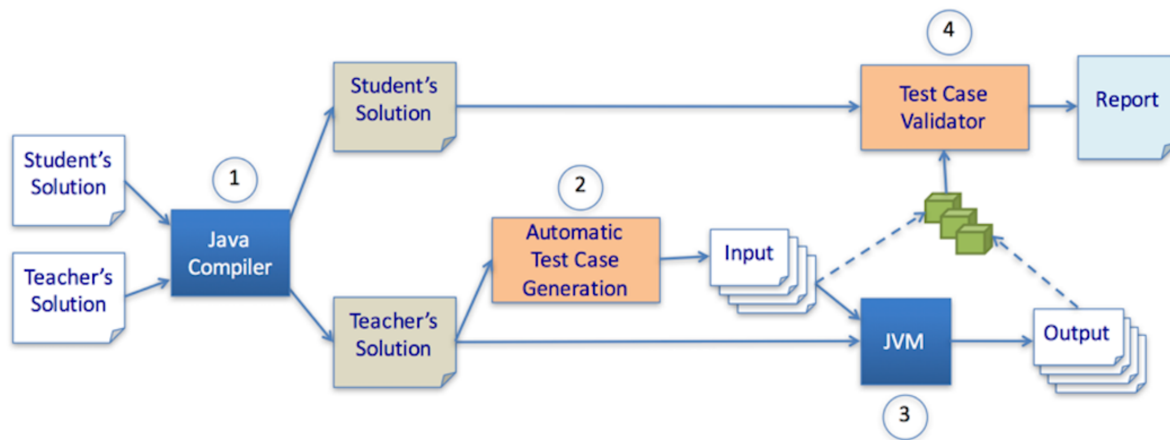


Fig. 8: Automatic test cases generation via source code static analysis

In step 2, we selected EvoSuite (Fraser and Arcuri, 2011) as the static code analyser because it does not generate test cases randomly. Contrarily, EvoSuite analyses the source code and for each condition it considers all possible condition values at least once, and for each program or subroutine, every entry point is called at least once. In other words, every branch (condition) is tested in both possibilities, true and false. Hence, tests cases created by EvoSuite try to maximise branch coverage, which is more exhaustive than statement coverage.

3.2.4 The assessment process

The data flow diagram (DFD) in Figure 9 summarizes the assessment process and clarifies how the triple report (compilation-analysis-testing errors) is built. The red boxes (`Teacher's Solution`, `Students' Solutions`, and `Self-Assessment Exercise`) represent the inputs given by the user. They correspond to those in Figure 7 (the teacher's solution is included inside a self-assessment exercise). There are four big grey boxes, which correspond to the different phases of the assessment. The first top-left phase represents the automatic test case generation from the teacher's solution during the construction time (see Figure 8). The other three phases (Phase 1, Phase 2, and Phase 3) correspond to compilation, analysis, and testing phases sequentially executed at assessment time (see Section 3.2.3). During the process, the phases are traversed starting at the big black bullet and following the paths of the DFD. Each phase produces reports with the information provided by the teacher. For this reason, there are dashed arrows from the `Ask the teacher` ellipses to a collection of

reports. These reports are finally combined to produce the final mark and report (indicated with a double-lined ellipse).

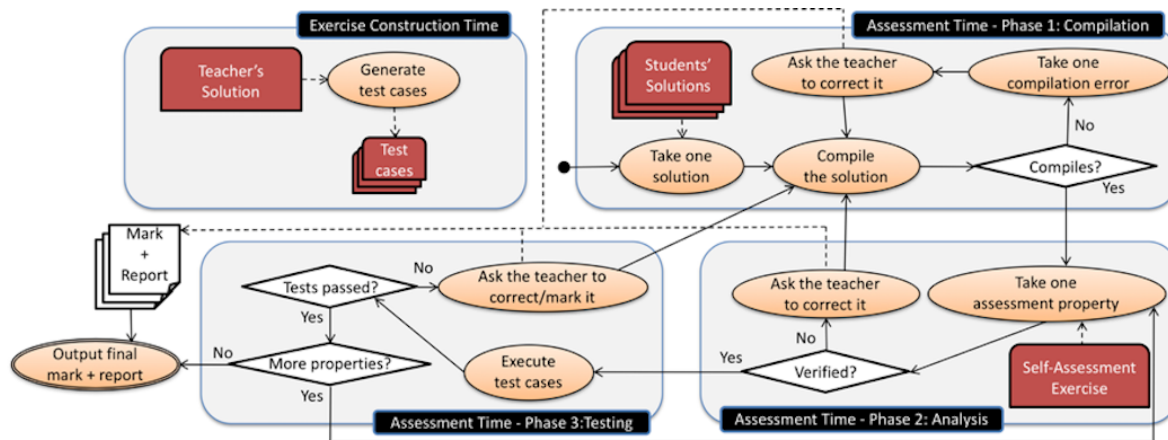


Fig. 9: Data Flow Diagram of ASys

It could be thought that the three phases are sequentially executed once, i.e., the solution of the student is compiled, afterwards, the generated code is analysed, and, eventually, the code is tested in order to detect runtime errors. However, according to Proposition 1, every time we find an error it must be corrected before assessing the rest of the code (otherwise this error may produce or hide other errors). Therefore, these three phases are repeated each time the teacher modifies the student's code; and this occurs every time a compilation error, property unsatisfied error, or failing test error is corrected. Hence, the general design of the DFD is: `error found` → `prompt the teacher` → `recompile the whole solution` → `check *all* properties` → `check *all* tests`. This sequence is repeated until no more errors are found. This scheme:

– allows for the correction (and marking) of any piece of code every time an error is spotted. With this method, teachers can correct an isolated error, several errors, or even the whole solution at once. Then, ASys checks again the whole program looking for new errors.

– avoids the introduction of errors by the teacher when correcting the code. For instance, when a teacher has previously corrected a property A, the correction of another property B could introduce an incoherence, making property A to fail again. In ASys, this fact is not a problem, because it recompiles the whole code and all properties are verified again.

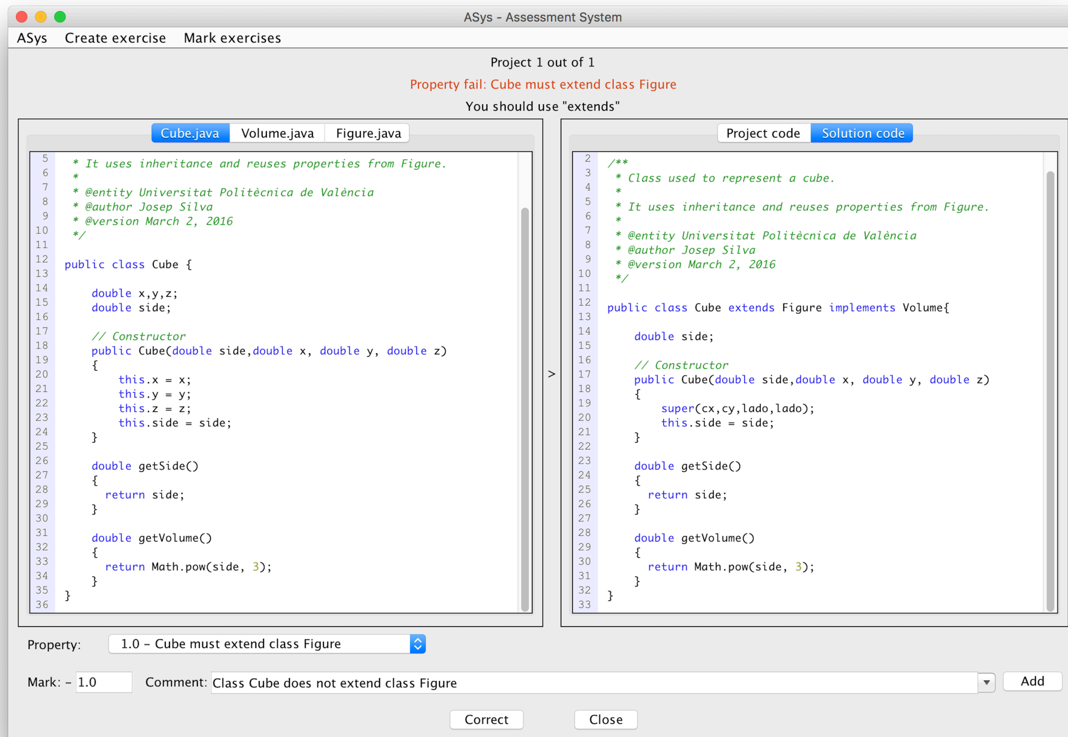


Fig. 10: Screenshot of ASys: marking an error at assessment time

Figure 10 shows a screenshot of ASys reporting an error: "*Cube must extend class Figure*". The left window shows the student's source code (in different tabs) where the cause of the error is identified automatically. To fix the problem, the user must modify this code. For this, in the right window, we can see the student's original source code, and the teacher's solution. Once the code has been corrected, at the bottom, there is a pane (only visible for teachers), where they can assign marks to the encountered problems, and optionally add comments, which will be included in the final report for the student providing explanations about the origin of the error. With button "Add", it is possible to add several marks and comments to a same property (e.g., because more than one problem is identified affecting this property). Note also that "Comment" is a list box. It contains all previous comments related to this property (normally done by the same or another teacher in another student's exam). This is particularly useful when correcting several solutions, because once a problem is identified and a mark/comment is introduced, it can be reused in different solutions and by different teachers. This feature considerably shortens the assessment time, because after assessing some solutions, the same errors

tend to be repeated over and over by (different) students; and the teacher only has to select a mark and a review that he has already introduced in the system.

4. EVALUATION

In this section, we present a real scenario of usage performed by several teachers (different form the authors) that took place at the *Universitat Politècnica de València* (UPV). We consider six Java exams of a second-year programming course where five teachers and 535 students participated.

The six exams were similar: class models composed of seven to nine classes (the statement of the problem, the resources given to the students, and the test cases and generated assessment code can be found at: http://personales.upv.es/josilga/ASys) . The assessment criteria were codified with 26-30 properties that were specified with the DSL as it was explained in Example 2 and illustrated in Figure 4. Then, the assessment template was automatically generated.

All exams were distributed among the teachers, who assessed them manually. Six months later, the same teachers assessed the exams again with ASys. ASys performed almost half of the assessment work automatically. ASys detected 100% of the errors and property violations. 48% of them were automatically corrected and marked by ASys (many students' exams were fully automatically marked, while others were only partially automatically marked). In the other 52% cases, ASys identified the error and showed the teacher's solution and the assessment criteria (thus speeding up the manual assessment), but the teacher was the one who finally decided the mark. The semi-automatic assessment was more accurate. As an average, the semi-automatic assessment mark was |0.49| over 10 points more accurate (the manual assessment errors where sometimes positive and sometimes negative).

Besides being exhaustive, the semi-automatic assessment was quicker and totally fair, applying the same criteria to all students and not being affected by psychological factors such as tiredness, mood, compensation, or also distorted view of an exam (as good/bad) after correcting a very bad/good exam.

With regards to the time gained with the automatic assessment, teachers invested one extra hour (as an average) in the preparation of the exam (instead of one, they used two hours). Therefore, the total time invested in the preparation was two hours. However, in the assessment of exams, half of the work was done automatically, and the other half was significantly faster because the detection of the error was automatic, the

exact file already opened and the line highlighted, and the teacher only had to correct it having the teacher's solution already opened, and in most cases, having available in a listbox the corrections already made for the same error of previous students. In total, the manual assessment invested 6 minutes per exam, while the automatic assessment invested 2.5 minutes (considering all exams, included those that were totally assessed by the tool). Therefore, the total time invested in the manual assessment was 6 minutes * 535 exams = 53.5 hours, while in the automatic assessment it was 2.5 minutes * 535 exams = 22.3 hours (31.2 hours less). Considering both the exam preparation and the assessment times, the automatic assessment saved 25.2 hours (47.1% of the total assessment time was saved).

This experiment allows us to approximate the relationship between manual assessment and ASys assessment. This relation is captured by the equation: $(2*P)+(N*0.42*A)$, being P the time needed to prepare an exam, N the number of student solutions to assess, and A the time needed to manually assess an individual student solution. Thus, the total time in the manual assessment is: $P+(N*A)$.

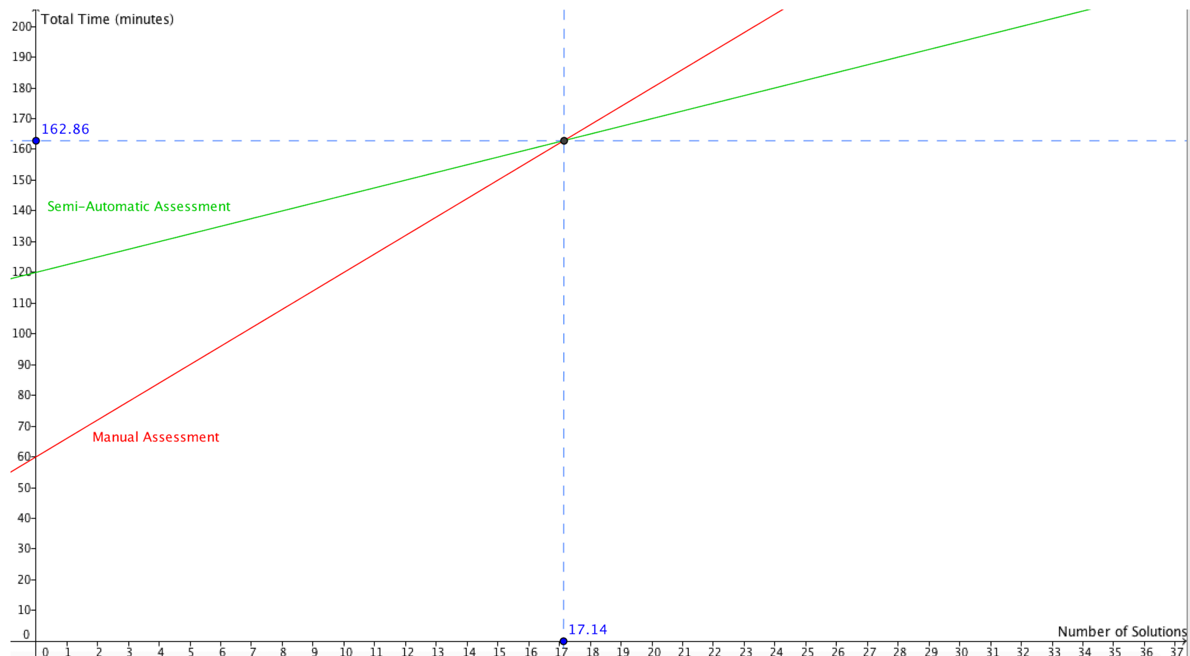


Fig. 11: Time needed to assess a number of student solutions (manual assessment vs ASys assessment)

Clearly, the time needed grows linearly with N, and thus for large courses the ASys assessment can save a lot of time. In our particular evaluation, the average value of A was 6 minutes and the average value of

P was 60 minutes. Therefore, the ASys assessment starts to save time with groups bigger than 17 students, as can be seen in Figure 11.

Additionally, there is an invaluable final advantage of the automatic assessment: the self-assessment exercises prepared remain after the assessment process as learning artefacts. They can be used by the students of the following courses to self-assess their work. When the use case was carried out, the students had 15 self-assessment artefacts (exams of previous courses) available.

In order to evaluate the platform and get feedback to improve ASys, a survey with 40 questions related to the use and utility of the tool was conducted. 164 students voluntarily filled the survey in. These 40 questions were divided into four different topics. The results of the survey are summarised in Figure 12.

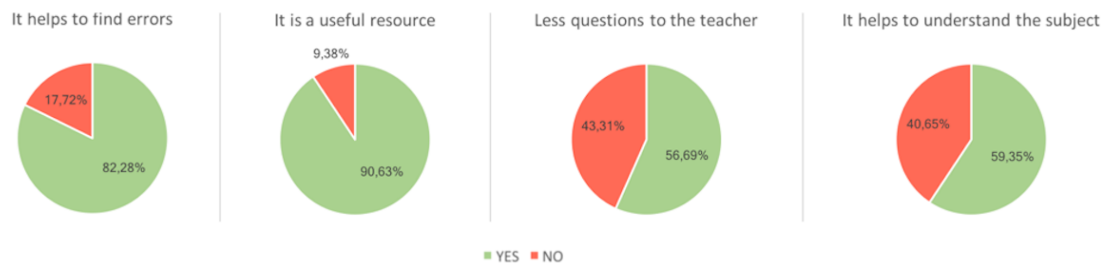


Fig. 12: Evaluation of ASys by 164 students

Finally, interviews with both the students and the professors were made. In general, the students found the tool very useful to self-evaluate their learning (in fact, they massively asked for a version for Haskell, because the course included Java and Haskell). The professors found the tool useful for three reasons: First, because it saved a lot of time in the assessment process. Second, because it increased the students' autonomy, and thus they received many less questions about how to solve the exercises. And third, because it increased the assessment quality, not only because it was exhaustive, but also because it was completely fair and equally applied to all students.

5. CONCLUSIONS

This work presents (i) a new methodology for the semi-automatic assessment of Java code and (ii) ASys, a new tool that implements this methodology. The new methodology proposes that exercises must be

executable artefacts. It conceives an assessment tool as an infinite loop that just executes self-assessing exercises. Each exercise is really a program that can analyse, compile, test, and change (correct) the source code of the student. This process is done by structuring the assessment in compositional properties.

ASys follows this approach using reflection. The reflective code is generated through the use of an assessment DSL with specific assessing abilities. This system has been evaluated with very good results in a study that took place at the UPV. More than 500 exams with a heterogeneous set of errors (including compilation errors) were assessed with the tool.

The use of AA tools is a good practice not only for automatically making part of the work, but also to increase the quality of the assessment; for instance, sorting the exams according to their type of errors, so that similar errors are corrected together (España et al., 2017). In this context, ASys allowed for identifying many assessment errors, and, simultaneously, it provided a systematic methodology to assess the exams, providing a fair assessment and also reports for the students and the teachers.

Currently, ASys has been officially integrated in all groups of four programming subjects (around 500 students per year) of two engineering degrees at the UPV. After our study, many students requested to have available more ASys' self-assessment exercises. We have developed a public small repository with ASys' self-assessment exercises about different aspects of Java programming (interfaces, abstract classes, inheritance, etc.).

Our experiments provide statistical data that approximate the relation between the manual assessment and the assessment made with ASys. In our particular case of study, where the manual average assessment time for an exercise was 6 minutes, and the time needed to prepare the exercise was 60 minutes, ASys saves time when there are more than seventeen student solutions to assess ($N > 17$). Despite of that, from our experience, even when $N < 18$ the use of ASys is definitely worth it because it increases the quality of the assessment with an exhaustive (reducing assessment errors) and fair (anonymous and objective assessment) analysis.

We have made our system and DSL publicly available and free, so that any researcher can use it or participate in their development. All the materials can be accessed at: http://personales.upv.es/josilga/ASys.

ACKNOWLEDGMENT

This work has been partially supported by MINECO/AEI/FEDER (EU), under grant TIN2016-76843-C4-1-R; PID2019-104735RB-C41, by the Generalitat Valenciana, under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Sergio Pérez was partially supported by Universitat Politècnica de València under FPI Grant PAID-01-18.

DATA AVAILABILITY STATEMENT

Data available on request from the authors. The data that support the findings of this study are available from the corresponding author upon reasonable request.

REFERENCES

Abd Rahman and Jan Nordin, 2007. Abd Rahman, K. and Jan Nordin, M. (2007). A review on the static analysis approach in the automated programming assessment systems. In National Conference on Programming 07.

Ahoniemi and Reinikainen, 2006. Ahoniemiand T. and Reinikainen T. (2006). ALOHA—A grading tool for semi-automatic assessment of mass programming courses. Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Department of Information Technology, Uppsala University, ACM, 2006, 139–140.

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

Annamaa et al., 2017. Annamaa, A., Suviste, R., and Vene, V. (2017). Comparing different styles of automated feedback for programming exercises. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research, Koli Calling ’17, pages 183–184, New York, NY, USA.

Benson, 1995. Benson M. (1995). Machine assisted marking of programming assignments, ACM SIGCSE Bull.

17 (1985), 24–25.

Bey et al., 2018. Bey, A., Jermann, P., and Dillenbourg, P. (2018). MA comparison between two automatic assessment approaches for programming: An empirical study on MOOCs. Journal of Educational Technology & Society, 21(2):259–272.

Brusilovsky and Sosnovsky, 2005. Brusilovsky P. and Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK.. ACM Journal of Educational Resources in Computing. 5. 10.1145/1163405.1163411

Bryan, 2019. Bryan, J.A. (2019). Automatic grading software for 2D CAD files. Comput Appl Eng Educ. 2019; 1-11. https://doi.org/10.1002/cae.22174

Buyrukoglu et al., 2019. Buyrukoglu, S, Batmaz, F, and Lock, R (2019). Improving marking efficiency for longer programming solutions based on a semi-automated assessment approach. Comput Appl Eng Educ. 2019; 27: 733– 743. https://doi.org/10.1002/cae.22094

Caballero et al., 2017. Caballero R., Riesco A., and Silva J. (2017). A Survey of Algorithmic Debugging. ACM Comput. Surv. 50, 4, Article 60 (November 2017), 35 pages. DOI:https://doi.org/10.1145/3106740

Caiza and Álamo Ramiro, 2013. Caiza, J.C. and Álamo Ramiro, J.M. (2013). Programming assignments automatic grading: Review of tools and implementations. In Proceedings of the 7th International Technology, Education and Development Conference (INTED2013), pages 5691–5700, Valencia, Spain.

Cedazo et al., 2015. Cedazo R., Garcia C., and Al-Hadithi B. 2015. A friendly online C compiler to improve programming skills based on student self-Assessment, Comput. Appl. Eng. Educ., 23, 887–896.

Chakraverty and Chakraborty, 2020. Chakraverty S and Chakraborty P. (2020). Tools and Techniques for Teaching Computer Programming: A Review. Journal of Educational Technology Systems. 49(2):170-198. doi:10.1177/0047239520926971

Daradoumis et al., 2019. Daradoumis, T., Marquès Puig, JM., Arguedas, M., and Calvet Liñan, L. (2019). Analyzing students' perceptions to improve the design of an automated assessment tool in online distributed programming, Computers & Education, Volume 128, 2019, Pages 159-170, ISSN 0360-1315. https://doi.org/10.1016/j.compedu.2018.09.021

Delgado and Medina, 2020. Delgado P. and Medina I. (2020). Customizable and scalable automated assessment of C/C++ programming assignments. Comput Appl Eng Educ. 2020; 1-18. https://doi.org/10.1002/cae.22317

Demir et al., 2010. Demir, Ö., Soysal, A. S., Arslan, A., Yürekli, Ö. Y. B., and lmazel (2010). Automatic grading system for programming homework. In Proceedings of the 2010 Computer Science Education: Innovation and Technology.

DeNero et al., 2017. DeNero, J., Sridhara, S., Pérez-Quiñones, M., Nayak, A., and Leong, B. (2017). Beyond autograding: Advances in student feedback platforms. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '17, pages 651–652, New York, NY, USA.

Denny et al. 2014. Denny P., Luxton-Reilly A., and Carpenter D. (2014). Enhancing syntax error messages appears ineffectual. In Proceedings of the 2014 conference on Innovation & technology in computer science education (ITiCSE '14). Association for Computing Machinery, New York, NY, USA, 273-278. DOI:https://doi.org/10.1145/2591708.2591748

Dogan and Bikbiyik, 2016. Dogan B. and Dikbiyik E. (2016). OPCOMITS: Developing an adaptive and intelligent web based educational system based on concept map model. Comput. Appl. Eng. Educ., 24, 676–691.

Edwards, 2003. Edwards S. H. (2003). Improving student performance by evaluating how well students test their own programs. J. Educ. Resour. Comput. 3, 3 (2003). https://doi.org/10.1145/1029994.1029995

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

España et al., 2017. España S., Insa D., Silva J. and Tamarit S. (2017). In what order should i correct the exercises? Determining the evaluation order for the automatic assessment of programming exercises. 16th International Conference on Information Technology Based Higher Education and Training (ITHET), Ohrid, 2017, pp. 1-3, doi: 10.1109/ITHET.2017.8067798.

Fisher, 1992. Fisher R. A. (1992). On the interpretation of $\chi 2$ from contingency tables, and the calculation of P, J. R. Stat. Soc. 85 (1992), 87–94.

Fraser and Arcuri, 2011. Fraser, G. and Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ’11, pages 416–419, New York, NY, USA.

Galan et al., 2019. Galan, D., Heradio, R., Vargas, H., Abad, I., and Cerrada, J. A. (2019). Automated Assessment of Computer Programming Practices: The 8-Years UNED Experience, in *IEEE Access*, vol. 7, pp. 130113-130119, 2019. doi: 10.1109/ACCESS.2019.2938391

Gavrilović et al., 2018. Gavrilović N, Arsić A, Domazet D, and Mishra A. (2018). Algorithm for adaptive learning process and improving learners’ skills in Java programming language. Comput Appl Eng Educ. 26: 1362– 1382. https://doi.org/10.1002/cae.22043

Georgouli and Guerreiro, 2010. Georgouli K. and Guerreiro P. (2010). Incorporating an automatic judge into blended learning programming activities, International Conference on Web-Based Learning, Springer, Berlin, Heidelberg, 2010, 81–90.

Gordillo, 2019. Gordillo, A. (2019). Effect of an Instructor-Centered Tool for Automatic Assessment of Programming Assignments on Students’ Perceptions and Performance. Sustainability 2019, 11, 5568.

Gross and Pickwart, 2015. Gross S. and Pinkwart N. (2015). Towards an Integrative Learning Environment for Java Programming. IEEE 15th International Conference on Advanced Learning Technologies, Hualien, 24-28, doi: 10.1109/ICALT.2015.75

Howatt, 1994. Howatt J. W. (1994). On criteria for grading student programs. SIGCSE Bull. 26, no. 3, 3-7. https://doi.org/10.1145/187387.187389

Ihantola et al., 2010. Ihantola, P., Ahoniemi, T., Karavirta, V., and 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, pages 86–93, New York, NY, USA.

Insa et al., 2015. Insa, D. and Silva, J. (2015). Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITICSE), pages 39-44.

Jackson, 2000. Jackson D. (2000). A semi-automated approach to online assessment. Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland. ACM, New York, NY, vol. 32, 2000, 164-167.

Jamil, 2017. Jamil, H. M. (2017). Automated personalized assessment of computational thinking MOOC assignments. In 2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT), pages 261–263.

Jovanovic and Jovanovic, 2015. Jovanovic, D. and Jovanovic, S. (2015). An adaptive e-learning system for Java programming course, based on Dokeos LE. Comput Appl Eng Educ, 23: 337-343. doi:10.1002/cae.21603

Joy et al., 2005. Joy M., Griffiths N., and Boyatt R. (2005). The BOSS online submission and assessment system. J. Educ. Resour. Comput. 5, 3, September 2005. https://doi.org/10.1145/1163405.1163407

Kelly and Tangney, 2006. Kelly D. and Tangney B. (2006). Adapting to intelligent profile in an adaptive educational system. Interact. Comput., 18, 385–409.

Király et al., 2017. Király S., Nehéz K., and Hornyak O. (2017). Some aspects of grading Java code submissions in MOOCs. Research in Learning Technology. 27. 10.25304/rlt.v25.1945

Liang et al., 2009. Liang, Y., Liu, Q., Xu, J., and Wang, D. (2009). The recent development of automated

programming assessment. In International Conference on Computational Intelligence and Software Engineering (CiSE 2009), pages 1–5. IEEE.

Liu et al., 2019. Liu, Xiao, Wang, Shuai, Wang, Pei and Wu, Dinghao, (2019). ICSE-SEET '19 Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training. Pages 126–137.

Marin et al., 2017. Marin, V. J., Pereira, T., Sridharan, S., and Rivero, C. R. (2017). Automated personalized feedback in introductory java programming moocs. In 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pages 1259–1270.

Muuli et al., 2017. Muuli, E., Papli, K., Tonisson, E., Lepp, M., Palts, T., Suviste, R., Sade, M., and Luik, P. (2017). Automatic assessment of programming assignments using image recognition. In Proceedings of the 2017 European Conference on Technology Enhanced Learning, volume 10474 of LNCS, pages 153–163. Springer.

Naik and Tripathy, 2008. Naik S. and Tripathy P. (2008). Software testing and quality assurance: Theory and practice, John Wiley & Sons, Hoboken, NJ, 2008.

Pettit et al., 2015. Pettit R. S., Homer J. D., Holcomb K. M., Simone N., and Mengel S. A. (2015). Are automated assessment tools helpful in programming courses? ASEE Annu. Conf. Expo., Seattle, WA, 2015. https://doi.org/10.18260/p.23569

Pieterse and Liebenberg, 2017. Pieterse, V. and Liebenberg, J. (2017). Automatic vs manual assessment of programming tasks. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research, Koli Calling '17, pages 193–194, New York, NY, USA.

Rahaman et al., 2019. Rahaman, Md. Afzalur and Latiful Hoque, Abu Sayed Md (2019). Automatic Evaluation of Programming Assignments Using Information Retrieval Techniques. Proceedings of International Conference on Computational Intelligence and Data Engineering, pages 47-57, Springer Singapore. ISBN 978-981-13-6459-4.

Restrepo-Calle et al., 2019. Restrepo-Calle, F, Ramírez Echeverry, JJ., González, FA (2019). Continuous assessment in a computer programming course supported by a software tool. Comput Appl Eng Educ. 2019; 27: 80–89. https://doi.org/10.1002/cae.22058.

Shamsi and Elnagar, 2012. Shamsi F. and Elnagar A. (2012). An Intelligent Assessment Tool for Students' Java Submissions in Introductory Programming Courses. Journal of Intelligent Learning Systems and Applications, Vol. 4 No. 1, 2012, pp. 59-69. doi: 10.4236/jilsa.2012.41006

Shao et al., 2019. Shao T., Kuang Y., Huang Y., and Quan Y. (2019). PAAA: An Implementation of Programming Assignments Automatic Assessing System. In Proceedings of the 2019 4th International Conference on Distance Education and Learning (ICDEL 2019). ACM, New York, USA, pp 68–72. https://doi.org/10.1145/3338147.3338187.

Skalka et al., 2019. Skalka, J., Drlík, M., and Obonya, J. (2019). Automated Assessment in Learning and Teaching Programming Languages using Virtual Learning Environment, *2019 IEEE Global Engineering Education Conference (EDUCON)*, Dubai, United Arab Emirates, 2019, pp. 689–697. doi: 10.1109/EDUCON.2019.8725127.

Souza et al., 2011. Souza D. M., Maldonado J. C. and Barbosa E. F. (2011). ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T), Honolulu, HI, 2011, 1-10. doi:10.1109/CSEET.2011.5876088

Souza et al., 2016. Souza D. M., Felizardo K. R. and Barbosa E. F. (2016). A Systematic Literature Review of Assessment Tools for Programming Assignments. 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET), Dallas, TX, 2016, pp. 147-156, doi: 10.1109/CSEET.2016.48.

Striewe et al. 2009. Striewe M., Balz M., and Goedicke M. (2009). A flexible and modular software architecture for computer aided assessments and automated marking. Proc. 1st Int. Conf. Comput. Supported Educ., Lisboa, Portugal, vol. 2, 2009, 54-61.

Subedhaa and Sridharb, 2012. Subedhaa V. and Sridharb S. (2012). A Systematic Review of Reusability Assessment Model and Related Approach for Reusable Component Mining. Journal of Computer Applications (JCA) ISSN: 0974-1925, Volume V, Issue 2.

Tung et al., 2013. Tung, S.-H., Lin, T.-T., and Lin, Y.-H. (2013). An exercise management system for teaching programming. Journal of Software, 8(7):1718–1725.

Wang et al., 2011. Wang T., Su X., Ma P., Wang Y. and Wang K. (2011). Ability-training-oriented automated assessment in introductory programming course. Comput. Educ. 56, 220–226.

Xiong and Suen, 2018. Xiong, Y. and Suen, H. K. (2018). Assessment approaches in massive open online courses: Possibilities, challenges and future directions. International Review of Education, 64(2): 241–263.

Yusof et al., 2012. Yusof N., Zin N. A. M., Adnan N. S. (2012). Java Programming Assessment Tool for Assignment Module in Moodle E-learning System. Procedia - Social and Behavioral Sciences, Vol. 56, 2012, 767-773, ISSN 1877-0428, https://doi.org/10.1016/j.sbspro.2012.09.714