



Automatic assessment of object oriented programming assignments with unit testing in Python and a real case assignment

Laura Climent  | Alejandro Arbelaez 

Departamento de Ingeniería Informática,
Universidad Autónoma de Madrid
(UAM), Madrid, Spain

Correspondence

Laura Climent, Departamento de
Ingeniería Informática, Universidad
Autónoma de Madrid (UAM), Madrid,
Spain.

Email: laura.climent@uam.com

Abstract

In this paper, we focus on developing automatic assessment (AA) for a topic that has some difficulties in its practical assessment: object oriented programming (OOP). For evaluating that the OOP principles have been correctly applied to a real application, we use unit testing. In this paper, we focus on prioritizing that the students understand and apply correctly complex OOP principles and that they design properly the classes (including their relationships). In addition, we focus on the Python programming language rather than the typical previous works' focus in this area. Thus, we present a real case study of a practical assignment, in which the students have to implement characters for a video game. This assignment has the particularities and advantages that it is incremental and that it applies all four OOP principles within a single assignment. We also present its solution with the UML class diagram description. Furthermore, we provide unit testing for this case study and give general advice for generalizing the unit tests to other real case scenarios. Finally, we corroborate the effectiveness of our approach with positive student evaluations.

KEYWORDS

automatic assessment, object oriented programming, python programming language, unit testing

1 | INTRODUCTION

In this section first, we describe the background of the paper, then we explain the literature review and finally, we describe the learning outcomes related to the object oriented programming (OOP) principles.

1.1 | Background

In recent years, teaching has been oriented toward continuous evaluation. For engineering degree programs, having several programming assignments is essential for the learning effect since the concepts acquired in the lectures can be applied to real

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2023 The Authors. *Computer Applications in Engineering Education* published by Wiley Periodicals LLC.

scenarios and the students will get feedback on the correctness of their code.

The teachers can evaluate the programming assessments either manually or (semi-) automatically. Providing several assessments manually during the whole course is a very demanding and time-consuming task, especially when the number of students is large. Lately, the number of students in engineering degree programs has tended to grow. In addition, there is another growing tendency for the high demand for Massive Open Online Courses (MOOCs). In these scenarios, grading manually several continuous assessments are infeasible or inadvisable given the operational scale. For these reasons, there is an increasing tendency for automatic assessments (AA).

Furthermore, AA has other advantages, such as the instant feedback that can be provided to the students, similar to one-to-one tutoring (which as motivated above, is infeasible for large-scale groups). AA can be oriented toward the learning outcomes of the course, allowing then, feedback to the teacher and external observers to check that the students have met the learning goals. Another beneficial aspect of AA is the lack of biased assessments. The assessment is a delicate task in teaching because involves a certain grade of subjectivity. Furthermore, this level of subjectiveness can give rise to possible informal/formal complaints from the students.

The literature review of AA of introductory programming modules/topics is extensive. However, when the teaching of programming is intermediate/advanced, there are more difficulties in the design and implementation of AA. In this paper, we focus on the AA of OOP. The difficulty of teaching OOP was mentioned, among others, in [1], [2], and [3]. In addition, the authors of [4] mention that deciding how to assess each student's OOP programming skill is one of the biggest challenges for educators who teach such programming courses. This is supported by a survey done among educators. Evaluating free answers provided by the students to theoretical questions does not fulfill all the learning outcomes of OOP. Then, it is not useful for assessing how to apply such theoretical concepts to real programming problems.

The above-mentioned reasons have motivated the work developed in this paper by going one step further and presenting an AA approach for OOP with unit testing that focuses on assessing the correct understating and application of (complex) OOP principles to real coding problems. In addition, we present a real case scenario, which is “the implementation of characters for a video game” and it has the particularities and advantages that it is incremental and that it applies all four OOP principles to solve a single scenario (these particularities are desirable, as we will mention in Section 2).

We use the Python programming language for both, the AA approach (with the *unittest* library) and the case study. As mentioned in [5], python is being credited as the fastest-growing programming language in recent times. This includes its use in the object-oriented programming paradigm. However, many previous works on unit testing for OOP are developed in Java language. While there are previous works of unit testing for OOP in the industry with Python, we are not aware of their existence for AA in teaching. This gives an extra novelty value to our work presented in this paper.

There are only two previous works on AA with unit testing for OOP oriented in teaching (but developed with Java): [6] and [7] (described in detail in Section 1.2). While they AA some of the OOP principles, we also include the AA of more complex OOP principles and the class design (including their relationship). Overall, the main contributions of the paper over the two previous works are:

1. AA of the inheritance between the classes provided in the assignment (including a case study with three levels of inheritance).
2. AA of the creation of the parent classes (including a method that works for whatever name that the students assign to the classes).
3. AA of aggregation between classes.
4. AA of the overriding of specialized methods.
5. Incremental assignment in which the software requirements evolve in two stages, in the same way as the AA feedback does.
6. AA of overload of operators.
7. AA of the generalization of methods that contain minor variations for child classes (therefore, the students have to check the type of instances).
8. AA of the Python `__str__` method (special method from Python that is used for displaying the information of an object).
9. AA of error handling in Python. Including the wrong manipulation of protected attributes.

Furthermore, we present the solution of the real-case assignment as a detailed description of the UML class diagram. As mentioned, we provide unit testing in Python for this case study and give general advice for generalizing the unit tests for other real case scenarios.

We apply the approach presented in this paper to a course called Intermediate Programming, where more than half of the content is about OOP. This course is mandatory for all second-year BSc. in Computer Science and BSc. in Data Science and Analytics students at the University College Cork (located in Ireland). Furthermore, the content presented in this paper can be applied

to similar courses of other BSc. engineering degree programs, such as BSc. in Telecommunications, BSc. in Electronics, and so on.

A comparison of the results that are achieved by the AA approach presented with the evaluation done by visual inspection of the student's code confirms the effectiveness of the designed approach. In addition, we also received positive student evaluations about this automatic assessment and the specific case study proposed.

1.2 | Literature review: Unit testing for OOP in learning

In this section, we leave three subareas out of the literature review: (i) AA of programming assignments (without unit testing and not for OOP), (ii) unit testing in learning (not for OOP), and (iii) unit testing for OOP in the industry. Regarding these three subareas, we would like to mention that [8] presents a detailed review of works in subarea (i). About (ii), there are many works of unit testing for learning introductory programming courses (e.g., [9], [10], and [11]). Regarding (iii), among others, some authors apply ML to OOP (e.g., [12]), while other works focus on automatically generating test cases for unit testing (e.g., [13], [14] and [15]).

We are only aware of two similar previous works on AA with unit testing for OOP in teaching (but in Java): [6] and [7] (the same author has coauthored both papers). In both works, the students are provided with wrapper classes in Java language with the definitions of all the functions that the assignment requires, for example, `getPerson()`, and exceptions, for example, `ErrorNonexistingPerson`. Furthermore, the authors provide unit tests in JUnit format for checking that every function from the wrapper class has been correctly implemented by the students.

The main disadvantage of the two previous works, concerning the OOP principles, is that the students cannot be asked to design a class diagram. Then, they do not evaluate complex OOP principles, such as inheritance, overriding of specialized methods, generalization of methods, overload of operators, and aggregation/composition. This is because the students have been provided with the class wrappers and with the definitions of the functions. However, as mentioned in Section 1.1, in this paper, we present an AA approach and case study that are more complete and evaluate all these above-mentioned complex OOP principles.

The other important difference between the work presented in this paper in comparison with the other two previous works is that we develop both, the real case

assignment and the AA tool in Python. Instead, the previous works typically use Java language. This represents a contribution to the education sector, specifically in the OOP modules. In addition, we present an incremental case study assignment (while the other two previous works just present a single assignment). This incremental assignment has the advantage of providing AA feedback to the students incrementally, which improves the learning process.

We would like to highlight that assessing all these above-mentioned complex concepts, entail more complex methods rather than providing the students with the wrapper classes. This represents a challenge and an important contribution to this paper. The rest of the paper provides details about how this has been achieved.

1.3 | OOP principles and learning outcomes (LOs)

In this section, we describe how the LOs of an OOP module (such as the course evaluated in this paper) are related to the OOP principles. Following, we enumerate the OOP principles.

1. *Encapsulation* consists of the bundling of related data and methods into a single entity (the class). Encapsulation also allows the code to be loosely bound. Designing correctly the classes especially, in a project that grows over time (in the same way as occurs in Section 3.2) is a LO. Encapsulation allows the modification of the data only via the methods, which makes the code more robust. Then, another LO is to properly control how and when the attributes are changed and catch erroneous uses before it has a serious impact.
2. *Abstraction* allows the hiding of members, methods, and implementation inside the class. There are three keywords associated with how hidden are the class members:
 - If the class member is declared as *public* then it can be accessed everywhere.
 - If the class member is declared as *protected* then it can be accessed only within the class itself and by inheriting child classes.
 - If the class member is declared as *private* then it may only be accessed by the class that defines the member.

The ability to categorize properly the class members is a LO.

3. *Inheritance* is a very useful principle of OOP that builds relationships between classes. This allows them to share class members and methods. Inheritance

captures an “is-a” relationship between classes. It allows one to take an existing class and specialize it and/or extend it. Identifying these inheritance relationships, which entail reusing the code is a LO. There is also a similar concept called *aggregation/composition*, which captures a “has-a” relationship between classes. Then several subclasses can compose another class and the subclasses might be reused later in different parts of the implementation. The main difference between aggregation and composition is that composition implies a strong dependence between the classes. That is, the contained class will be obliterated when the container is destroyed. While, with aggregation, the contained class will remain even when the container is dissolved. A LO is to correctly differentiate between all these relationships and apply them correctly.

4. *Polymorphism* means that a method can cope with different types of inputs. Then, the same code can be applied to multiple data types. There are two types of polymorphism: overloading and overriding.
 - *Overloading* allows a class to have multiple methods with the same name but a different set of parameters and implementation. A LO is to code the overloading of operators. For instance, a very popular one is the + operator (e.g., for concatenating two strings).
 - *Overriding* occurs when replacing an inherited method with another having the same signature. Customizing the behavior of subclasses by using this mechanism is a LO.

OOP design involves the use and application of the above-mentioned OOP principles for the design of a solution for coding problems. One of the most popular modeling approaches is the Unified Modeling Language (UML) class diagram, which offers a view of the classes, their attributes and methods, and the relationships between them. The UML diagram offers a visual representation that is independent of the implementation, coding language, and so on. An example of a UML class diagram of a case study is represented in Figure 1 (in Section 3). Designing properly the UML diagram is a LO of many OOP courses (such as the course evaluated in this paper).

2 | OOP TEACHING AND ASSESSMENT

In this section, we discuss typical methods of OOP teaching and assessment, highlighting their advantages and disadvantages.

When teaching OOP, the visualization of the different components plays an important role. For this reason, typically UML class diagrams are used. Students typically do not find it difficult to learn the theory behind these OOP concepts and principles. However, when it comes to applying such a theory to solving real-world problems, the students struggle. Traditional approaches to the assessment of OOP consist in assigning numerical scores to theoretical questions or specific coding questions.

These approaches do not reveal to the students how they can apply their knowledge of the OOP principles to solving real programming problems. Therefore, there is a motivation for creating other ways of assessing and providing feedback to the students that allow them to apply OOP effectively to real applications. In [16], the authors mention the need/aim of making a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding, all together.

There are other types of approaches for the assessment of OOP that involve coding exercises. Some of them use a single coding exercise for the demonstration of the application of a single principle. Typically, students can solve these short problems without difficulty. Nevertheless, in this assessment scenario, there is still a lack of connection between the four OOP principles (encapsulation, abstraction, inheritance, and polymorphism) and how to apply them to a more complex real-life application. This represents a typical struggle for students. However, this type of assessment is more similar to the real OOP applications that students would develop in the industry. For this reason, in the next section, we present a real OOP application case study—“the implementation of characters for a video game”—that applies all four OOP principles to solve this single scenario.

The last aspect to consider is the fact that it is acknowledged that learning programming skills are a step-by-step procedure. Feedback during the solving process provides a substantial improvement in the programming solution. This incremental solving process implies that students get AA feedback about the errors and consequently, they can discover better approaches to deal with the following parts of the assignment. For this reason, authors of papers, such as [17], state that if the running example takes several weeks to cover in class, the students should be given more than one assignment to grow the program in this period. This has been one of the motivations for creating a case study divided into two parts (Sections 3.1 and 3.2). Where the feedback about the first part helps students not only to learn about their mistakes but also to develop the second part of the assessment.

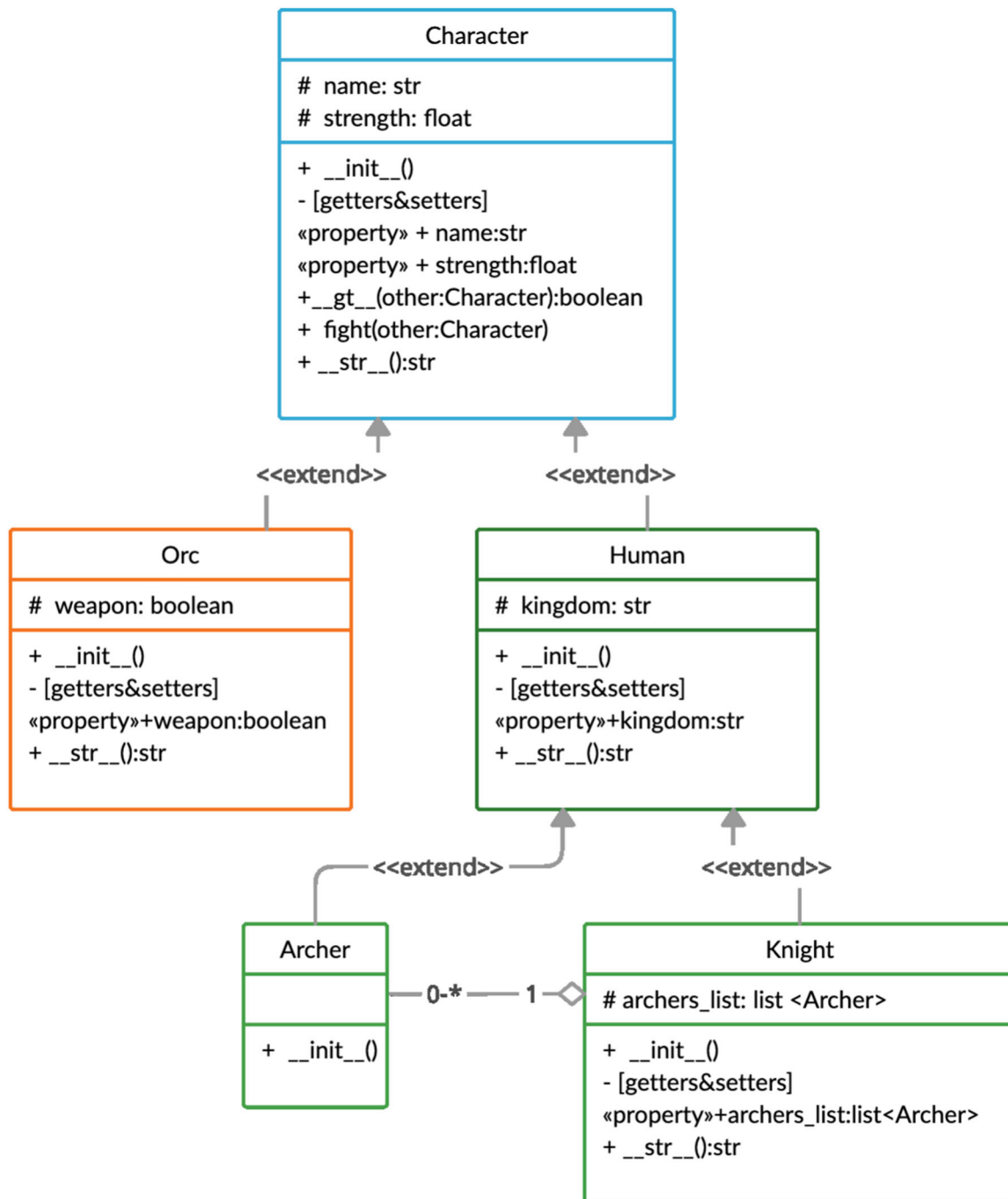


FIGURE 1 UML class diagram.

3 | REAL CODING CASE STUDY

In this section, we present a case study scenario that considers all four OOP principles (Section 1.3) in the same unique coding problem. Other coding problems of different domains can be used. However, to fulfill all the learning outcomes, they should include all the OOP principles. In Section 4 we show the solution of the assignment with a UML diagram. In addition, in Section 5 we show how to automatically assess such

principles with unit tests. The authors of [2] also present a unique problem that focuses on combining the OOP principles. However, unlike our case study, it does not consider the overloading of operators and error handling for data manipulation.

We present a real coding case study that is oriented toward the topic of video games. Other authors, such as in [1] presented an assignment combined with a video game and they justify that there is a pedagogical reason behind this choice. First, nearly all the students enjoy

computer games and therefore it is funnier for them to work on them than with more conventional types of projects. Second, computer games are composed of a certain number of interacting complex objects that can be modeled with OOP design.

The case study scenario and the unit tests that we present in this paper are implemented in Python. The motivation for the selection of this language can be found in Section 1.1. The assignment consists of developing a *library application that allows the creation of several characters of a battle video game. The characters have certain types of characteristics and functionalities, such as fighting between them.*

The case study coding problem presented to the students is divided into two stages and presented to the students in an incremental way. First, a small subpart, and later, once this subpart has been completed, the whole problem is released. One of the reasons behind this decision is that the students can get familiar with the real problems in the industry, for which the requirements of the software to be developed change over time due to changes in the needs of the clients. (Who typically do not know software development and therefore struggle to have a clear and specific definition of what they want). In these situations, thanks to the right use of OOP principles, the code is loosely bounded (see the OOP principle encapsulation in Section 1.3) and it can be reused and adapted easily. As previously mentioned, the other reason behind this choice is that the students can get feedback about the progress of their assignment in several stages.

3.1 | First part of the assignment: A single class

The first part of the assignment consists of creating the class *Orc*, which is defined by the following attributes:

1. The *orc's name* (e.g., "Ogrorg")
2. The *orc's strength* score in the domain [0-5]
3. Does the orc have a *weapon?* within the domain [True, False]

The methods associated to the class are:

1. A *constructor* for initializing instances (e.g., `orc1 = Orc("Ogrorg", 4.3, True)`). In Python, the constructor is a special method called `__init__`.
2. The *properties* for accessing and modifying the values of all the attributes (e.g., `orc1.name = "Grunghi"`).
3. The `__str__` method, which is a special method from Python that is used for displaying the information of

an object in a nicely formatted string representation. It should return the string representation of the instances in this format: `name strength weapon` (e.g., `Orc("Ogrorg", 4.3, True)`).

4. The overload of the `>` *operator* determines if an orc is greater than another in terms of fighting. If an orc owns a weapon and the other one does not own it, then the orc with a weapon is greater than the other one. If both orcs are in the same situation of ownership of weapons, then an orc is greater than another one only if its strength is greater (e.g., `orc1 > orc2`).
5. The orc has a functionality, which is the *fight method* with another orc. If an orc is than another orc, then it wins, and its strength increases by 1 point. Besides, the instance of the winner orc is printed on the screen. Otherwise (there does not exist a `>` orc), they both lose 0.5 points.

Furthermore, the students are asked to implement types and values *errors handling* for the attributes of the class. They must check that the values' setting by external users of the software is correct and within the domain of the attributes. The assignment specifies that for numeric values if the user introduces a greater value than allowed, it will be truncated to its max. value (e.g., the max. strength value is 5). If the value is lower than allowed, then it will be truncated to the min. value (e.g., the min. strength value is 0). For errors related to the type of the attributes (e.g., trying to assign strength equal to "Ogrorg"), the assignment will not be completed and the error message "type ERROR" will be printed on the screen.

In addition, the students have to implement a test module for checking all the functionalities of the *Orc* class. They have to import the module *Orc* and test each of the above-described functionalities.

3.2 | Second part of the assignment: Combining several classes

In the second part of the assignment, the students are told that the clients of the video games company for which they work (hypothetically) tell them that they realized later on that more characters have to be implemented. Then, they have to use advantageously the OOP principles (Section 1.3) for coding reuse. The new characters are humans: archers and knights.

The classes *Archer* and *Knight* have both the attributes (1) name and (2) strength (such as *Orcs*, see their description in Section 3.2). Besides, they have some extra attributes:

3. Their *kingdom* (e.g., “Gondor”).
4. Only the knights have an extra attribute: *archer_list*, a list of objects of type Archer. This list represents the archers that are led by the knight. There is no limit to the number of archers that a knight can lead. However, they all must belong to the same kingdom as the leader Knight. If the list contains archer/s that belong to another kingdom, they will be removed from the list and only the other archers who satisfy such constraint will be assigned to the knight. The error message “archers list ERROR” will be printed on the screen.

The archers and knights have the same methods as the orcs. However, there are some differences:

1. All the characters (archers, orcs, and knights) must be able to use the `>` operator with any other type of character. Note that archers and knights do not have the attribute `weapon`.
2. Archers and knights can only fight with orcs. The error message “fight ERROR” will be printed on the screen if they try to fight against each other. Apart from this, the functionality `fight` works in the same way as the orcs.
3. The `__str__` method returns the string representation of the instances in this format: `name strength kingdom` (e.g., `Aragorn 3.3 Gondor`). Besides, only for the knights, the list of archers that leads will be shown right after within square brackets (e.g., `Aragorn 3.3 Gondor [archer1 2.1 Gondor, archer2 4.2 Gondor]`).

The students have to extend the implementation of the types and values errors handling for the attributes of the new classes (in the same way as described in Section 3.1). Furthermore, they have to add new errors handling mentioned above for the archers and knights. The students also have to extend the test module from the previous section, so that it is not only checking the functionalities of the orcs but also the functionalities of the archers and knights.

4 | SOLUTION AND UML CLASS DIAGRAM

The first task that the students have to do after the release of the statement of the second part of the assignment, is the design of the UML class diagram. Then, once it is completed, they can continue with the implementation. The students are reminded that they have to use the OOP principles (Section 1.3) and that

marks depend on the right use of such principles. Therefore, they must reuse the code by using the inheritance OOP principle rather than copying and pasting code. The first step for the UML class diagram design is to analyze the description of the application and decide on the candidate classes and their relationship to draw the overall class diagram. As mentioned in [2], in this process the students will apply all four OOP principles to come up with a good and clean class diagram, that is, tightly encapsulated, loosely bounded, and highly cohesive.

4.1 | Encapsulation and inheritance

In this case study, the classes candidates that can be easily seen are Orc, Archer, and Knight. In fact, we provide the names of these classes to the students, specifying that they must keep the right spelling. This is very important for the future use of the unit tests. In addition, it is easy to observe one common property (`kingdom`) and common specialized methods (`>` and `fight`) among the classes Archer and Knight. For this reason, by applying the OOP principle of inheritance, the students should be able to realize that there is a superclass for both classes (in this paper, we call it *Human*). The other inheritance is, perhaps, not as easy to realize. All the classes have two common properties (`name` and `strength`) and three common methods (`>`, `fight`, and `__str__`) (even if there are specializations). Then, there should be a superclass of all the classes (in this paper, we call it *Character*). Figure 1 shows these two inheritance relationships between the five classes mentioned. We would like to mention, that as we will explain in more detail in Section 5, one of the advantages of our AA approach is that we do not need to specify to the students how many parent classes there are for the characters, neither their names nor their distribution. This represents the main advantage since the students have to think about how to design the inheritance to maximize code reuse. In addition, the lecturer can automatically assess if the inheritance has been applied properly.

4.2 | Composition/Aggregation

To take advantage of the OOP principles, students should go one step further by realizing a has-to relationship between a knight and a set of archers. A knight leads a set of archers that belong to the same kingdom. Here, it is important noticing that the knights do not have a strong dependence on the archers. A knight instance can exist

even when its archers are deleted. For this reason, the relationship between these classes is an aggregation rather than a composition. Recall that the aggregation is represented as a white diamond. In addition, the UML diagram (Figure 1) should include the numbers 1.* (the asterisk means more than 1) archers led by 1 knight.

4.3 | Abstraction

The UML of Figure 1 should also reflect the abstraction principle. For this reason, the attributes of the classes (name, strength, weapon, and kingdom) are preceded by the # symbol. We define them as protected rather than private because we want that they can be accessed not only within the class itself but also by inheriting child classes. To access attributes, we use properties. A property provides a flexible mechanism to read or write the value of a private/protected field. The syntax for reading and writing of properties is the same as the fields (e.g., for the field weapon, the property is called weapon). But properties are typically translated to “getters” and “setters” method calls. This allows the data validation and error handling associated with the fields. We define the properties as public (symbol +) and the getters and setters as private (symbol –), therefore, the fields must be accessed by using the properties. The rest of the methods (constructor, fight, the > operator, and __str__) are declared as public (symbol +) so that they can be accessed outside the class. That is to say, by external software modules that include the classes of this assignment.

4.4 | Polymorphism

This assignment also considers the two types of polymorphism: overloading and overriding. It includes the overloading of the > operator and the overriding of specialized methods that have been inherited.

4.4.1 | Overloading

In this assignment, the students have to overload the > operator, which is specified in Python with the special method `__gt__`. The definition of the > operator involves the use of the attribute weapon for the characters that have this attribute. Note that only orcs have it, while humans do not. For this reason, this method, which belongs to the class Character (because is common to all the characters) has to differentiate if the instances are orcs or humans. This can be achieved by using a method that checks the type of the instance (e.g., `type(self)!=`

Orc). Then, different characters can interact with this operator, for example, `Aragorn > Ogrorg` would return true or false according to the definition of the operator.

4.4.2 | Overriding

To display the information of a character in an adequate format, the Character class has the special method `__str__` that prints its two attributes (e.g., `Ogrorg 4.3`). Then, this method is specialized in both classes Orc and Human. For reusing the code, in both classes, the method should call to the same method of the parent class (i.e., `super().__str__()`). Therefore, the specific method of Orc prints the value of the weapon (e.g., True) and the specific method of Human prints the kingdom (e.g., Gondor). In addition, the specific method of Knight prints the list of archers (e.g., [`archer1 2.1 Gondor, archer2 4.2 Gondor`]). Note that the Archer class does not have a `__str__` method, since it does not have any new attribute with respect Human. Therefore, when there is a call to this method for an archer, the method of the parent class (i.e., Human) will be executed.

At first sight, it could seem that the method fight has to be overridden for orcs and humans. However, the method fight has the same characteristics for both classes, with the exception that humans can not fight with humans. This check can be done in the same way as we did with the > operator, by checking the type of the instance. If none of the two instances of the fight are orcs, this implies an error (i.e., `type(self)!= Orc` and `type(other)!= Orc`). Therefore, as shown in Figure 1, fight is a common method for all the characters.

4.4.3 | Error handling

As previously mentioned, the encapsulation and abstraction principles from OOP allow making the code more robust because the software developer can code the error catching of erroneous uses of the attributes before they have a serious impact on the software usage. The error handling can be introduced in the part of code that controls the modification of the attributes. That is to say, in the setters associated with the attributes. Note that when the user uses the properties of the attributes, the properties call to the getters/setters and therefore the error handling also takes place. Specifically, for the case study presented, for each attribute (i.e., name, strength, kingdom and archers_list), in its setter function, the students must check that the type of the input parameter is correct (e.g., `type(name) == str`) and that its value is within the range (e.g., `5 >= strength >= 0`). Furthermore,

the functionalities of the classes can produce errors if applied in the wrong way. For this case study, there is only one error in the method associated to the function `fight` (as mentioned in Section 4.4.2) the fight of two humans is not allowed.

We would like to mention that error handling can be treated in several ways according to the knowledge of the students about it. The ideal situation would be to through and catch exceptions. Since this is not in the scope of this paper and in some OOP courses the students still do not have this knowledge, we did not include it. Therefore, the students are only asked to print on the screen an error message in a specific format (e.g., “archers list ERROR”). This can be modified by the lecturer when using our case study. For all the possible errors of the case study assignment and their associated error message on the screen, see Section 3.

5 | AA APPROACH WITH UNIT TESTING

In this section, we present the AA testing for the students' code for the case study assignment presented in Section 3. The main objective of unit testing is to ensure the correctness of a given unit of code by checking its correct functioning. As mentioned in [13], compared to numeric test data, test programs optimized for object-oriented unit testing are more complex. For this reason, method call sequences that realize interesting test scenarios must be evolved. During the execution of the testing, all the objects participating in the assignment should be created and put into particular states by calling several instance methods for these objects. Typically, each test case focuses on one particular method. To perform complete unit testing, all the methods within the class have to be checked.

As previously mentioned, without loss of generality, we have adopted Python as the coding language of the case study and the AA approach presented in this paper. Python offers a library for unit testing called `unittest` that we import into our test file. In addition, we import the code of every student and we name it `mod`. We also include the typical basic libraries, such as `io` and `sys`. Then,

after importing such libraries, we can create a new class (we call it `CharactersTest`) that inherits from `unittest` and includes all the tests. Following, the code associated.

```
import io
import sys
import unittest
import code_student as mod

class CharactersTest (unittest.TestCase):
    ...
```

Each unit test case consists of one method call sequence and one or more assertion statements [13]. The `assert` statement checks that the condition that follows is valid. If all the `assert` statements of a test are valid, the result of the unit testing for the test will be *ok*. Otherwise, it will be *fail*. Below there are two examples of the output of two unit tests, where the first one fails and the second is correct.

```
test_values_range_constr_orc
(__main__.CharactersTest) ... FAIL
test_str_Archer
(__main__.CharactersTest) ... ok
```

For calculating the marks of the assignment, every unit test can have a portion of marks associated. Therefore, the correction of the assignments can be done automatically by just running the unit tests presented in this section. In addition, we can change the verbosity of the unit tests, by including the following line of code in the main function.

```
unittest.main(verbosity=2)
```

Then, when a unit test fails, it will show details about the failing. For example, in Figure 2, the strength of the `orc2` should be zero but it is not. For this reason, the unit test failed.

In addition to the unit tests, we define an external function that is in charge of catching the standard output of the student's code. We need this function to check if the student's code is handling errors in the way specified

```
=====
FAIL: test_values_range_constructor_Orc (__main__.charactersTest)
-----
Traceback (most recent call last):
  File "/Users/lcliment/Desktop/pruebaPython/unittestcharacters.py", line 47, in test_values_range_constructor_Orc
    assert orc2.strength == 0 # test : the result should be: 0
AssertionError
-----
```

FIGURE 2 Unit test failing message.

by the assignment (with message errors printed on the screen, such as “type ERROR”). Once the standard output is caught and stored as a string, we can compare it with the correct error message. The function is called `capt_out()` and its definition and its associated imports are described in Appendix A.

Thereafter, we describe each type of unit test. To avoid repetition of similar code, we do not include all the unit tests for the three types of characters. Instead, we describe each type of unit test and then we include the rest of the code in Appendix A. For clarifying purposes, we also highlight in bold the important values under testing within each unit test.

5.1 | Constructors

In this section, we present the type of unit test associated with the attributes' update within the constructors: the types of values and the range of the values.

Regarding the range of values checking, the characters' strength can not exceed the limits of the allowed values [0,5]. In the code below we show such testing for orcs. We create an orc with strength 5.3. Then, we assert that the strength attribute has been truncated to the max. value (5). The same should happen with a lower strength than the min. allowed (0). Similar unit tests must be done for checking the archers and knights' strength values (see Appendix A.2).

```
def test_values_range_constr_orc(self):
    orc1 = mod.Orc("Ogrorg", 5.3, True)
    assert orc1.strength == 5
    orc2 = mod.Orc("Grunch", -100.0, False)
    assert orc2.strength == 0
```

Regarding the types of values checking, we have to ensure that when we create new characters with wrong types for their attributes, a “type ERROR” message appears on the screen. For example, the orcs' attributes are name, strength and weapon. For this reason, in the code below, we create three orcs, each with a wrong type for each attribute. Similar unit tests must be done for archers and knights (including checking their kingdom). Furthermore, for the knights, we have to check that their list of archers is composed of archers of their same kingdom. These unit tests for humans can be found in Appendix A.2.

```
def test_values_types_constr_orc(self):
    with capt_out() as (out, e):
        mod.Orc(1, 4.3, False)
    assert (out.getValue().strip() ==
           "type ERROR")
```

```
with capt_out() as (out, e):
    mod.Orc("Grunch", "Grunch", False)
    assert (out.getValue().strip() ==
           "type ERROR")
with capt_out() as (out, e):
    mod.Orc("Ogrorg", 4.3, "Ogrorg")
    assert (out.getValue().strip() ==
           "type ERROR")
```

5.2 | Properties

In this section, we present the type of unit test associated with the attributes' update within the properties. As within the constructor, we must check the types of values and the range of the values. Then, we have to ensure that after creating each of the characters, the retrieval and update of information on their attributes is correct.

For example, below we show the unit test that asserts that the kingdom of a knight coincides with the kingdom of the archers that he/she leads. First, we create two archers and one knight (with an empty list of archers). Then, we assert that indeed it is empty by using the property for getting the `archers_list` attribute.

Subsequently, we update this attribute to a list that contains these two archers previously created, by using its property. Since only the first archer belongs to the same kingdom as the knight, it should be the only one assigned (we include a second assert statement that ensures this). The last assert sentence checks that the values types of the `archers_list` attribute are correct. Otherwise (the list contains something different than archers, for example, in the code below appears an integer value), the “archers list ERROR” message is shown on the screen.

Similar unit tests must be done for checking the properties of the attributes of the characters: name, strength weapon and kingdom (see Appendix A.3).

```
def test_property_knight_archers_list(self):
    a1 = mod.Archer("a1", 3.3, "Gondor")
    a2 = mod.Archer("a2", 2.3, Rohan)
    k1 = mod.Knight("k1", 4.0, "Gondor", [])
    assert k1.archers_list == []
    k1.archers_list = [a1, a2]
    assert k1.archers_list == [a1]
    with capt_out() as (out, e):
        k1.archers_list = [a1, 3]
    output = out.getValue().strip()
    assert (output == "archers list ERROR")
```

5.3 | `__str__` method

A unit test for the `__str__` method has to be performed for every character. Here, we show the associated unit test for the archers, while the rest of the unit tests can be found in Appendix A.4. For this type of unit test, we have to ensure with an assert statement that when a character is printed on the screen, the correct data is displayed. Then, we just have to create a character with certain initial parameters and print it on the screen.

```
def test__str__archer(self):
    a1 = mod.Archer("a1", 3.3, "Gondor")
    with capt_out() as (out, e):
        print(a1)
    output = out.getValue().strip()
    assert (output == "Archer13.3Gondor")
```

5.4 | `>` operator

The next unit test is for checking that the overload of the `>` operator has been properly implemented. This operator can be applied to any of the characters. Here we present a unit test that combines its use between all the characters. First, we create an archer and a knight, where the knight is stronger. Then, we assert that the archer is not `>` than the knight. Subsequently, we create an orc that has a very low strength and we assert that he is not `>` than the knight or the archer. In Appendix A, specifically in Appendix A.5, we also show another unit test for the `>` operator with different orcs that have/do not have weapons.

```
def test_greater_archer_knight_orc
(self):
    a2 = mod.Archer("a2", 2.3, "Rohan")
    k1 = mod.Knight("k1", 4.0, "Gondor", [])
    assert not (a2 > k1)
    orc2 = mod.Orc("Grunch", 1.9, False)
    assert not (orc2 > k1)
    assert not (orc2 > a2)
```

5.5 | Fight method

The last functionality of the characters is fight. In addition, this method uses the previous overload of the `>` operator. Again, we create one character of each type and we make them fight between them, ensuring the correct functionality of such a method. Archers and

knight can not fight between them, therefore, the first two asserts statements check that the fight error messages are shown on the screen if these situations happen. We also create an orc with a weapon and make the humans fight against him. When the orc fights with the archer, the result is that the orc wins. For this reason, his strength is increased in one unit (his current strength is 5), while the strength of the archer remains the same. Subsequently, there is another fight between the orc and the knight. In this case, both have the same strength (5), therefore there is a match and both lose 0.5 points of their strength.

```
def test_fight_archer_knight_orc(self):
    a2 = mod.Archer("a2", 2.3, "Rohan")
    k1 = mod.Knight("k1", 5.0, "Gondor", [])
    with capt_out() as (out, e):
        a2.fight(k1)
    output = out.getValue().strip()
    assert (output == "fight ERROR")
    with capt_out() as (out, e):
        k1.fight(a2)
    output = out.getValue().strip()
    assert (output == "fight ERROR")
    orc1 = mod.Orc("Ogrorg", 4.0, True)
    a2.fight(orc1)
    assert (orc1.strength == 5 and
            a2.strength == 2.3)
    k1.fight(orc1)
    assert (orc1.strength == 4.5 and
            k1.strength == 4.5)
```

In Appendix A, specifically in Appendix A.6, we present an extra unit test that checks that the limits of the strength of the characters are correctly truncated after fighting.

5.6 | Inheritance between classes and overriding of methods

The AA checking of the right use of the inheritance between classes is quite difficult and it is a key point of the contributions of this paper. In the description of the assignment, we do not specify how the inheritance between classes should be (because figuring out such relationships is, itself, part of the assignment). We only mention that at least three classes must be created: Orc, Knight, and Archer. The classes Human and Character are the names that we used for the solution of the assignment presented. However, the students could have written different names for such superclasses or they could

not realize the existence of the classes (which implies a marks discount).

For this reason, we have to obtain the names of the superclasses for each student's assignment. For achieving this, we use the `__bases__` property of a class, which contains a list of all the base classes that the given class inherits.

In the unit test below, first, we define variables with the superclasses of each class. Then, we assert that Knight and Archer's superclasses are the same. In addition, their superclass has to be the same as the superclass of Orcs. Furthermore, we also assert that the superclass of Orc differs from the superclass of Archer and Knight. This is useful for scenarios in which the student's code only contains one inheritance, where all the characters inherit from the same superclass. Rather than the correct solution, which has another intermediate super class only for Archer and Knight.

```
def test_inheritance(self):
    super_Archer = mod.Archer.__bases__
    super_Knight = mod.Knight.__bases__
    super_orc = mod.Orc.__bases__
    super_Human = super_Archer[0].__bases__
    assert super_Archer == super_Knight
    assert super_Human == super_orc
    assert super_Archer != super_orc
    assert super_Knight != super_orc
```

In addition, we can use an object's attribute `__doc__`, which provides a documentation of the object. Specifically, it shows a message that says that a class has been defined as a subclass of another. For example, the message for Archer would be "Define Archer to be a subclass of Human." For checking the superclasses of the three classes, we just have to add the following code to the main function:

```
print(mod.Archer.__doc__)
print(mod.Knight.__doc__)
print(mod.Orc.__doc__)
```

Furthermore, we have to check that the overriding of the methods is correct. For instance, if the student uses inheritance but overrides all the methods from the superclass, the inheritance would be useless since there would not be code reuse. For achieving this, we use the attribute `__qualname__`, which means qualified name in Python. It gives you a dotted path to the name of the target object. When the attribute is applied to a method, it shows us in which class the method was defined. Then,

we can check if the method has been defined in the right class. We recall that Figure 1 shows the right methods definition by their generalization in superclasses. Since the `__qualname__` attribute is not applicable to the properties, we use the method `dir()`. This method returns the list of the attributes and methods of any object. Then, we can check that the getters and setters of the general attributes are defined in the corresponding superclasses. Below, the unit test associated:

```
def test_characters_overriding(self):
    assert mod.Orc.fight.__qualname__ != "Orc.fight"
    assert mod.Orc.__gt__.__qualname__ != "Orc.__gt__"
    assert mod.Archer.__str__.__qualname__ != "Archer.__str__"
    for i in [
        "__getName", "__setName",
        "__getStrenght", "__setStrenght",
        "__getKingdom", "__setKingdom"
    ]:
        assert "_Knight" + i not in dir(mod.Knight)
        assert "_orc" + i not in dir(mod.Orc)
        assert "_Archer" + i not in dir(mod.Archer)
```

In this unit test, the first two asserts test that the methods `fight` and `__gt__` have been inherited from the superclass of Orc. We ensure it by checking that they have not been defined in the class Orc. The third assert checks that the method `__str__` has not been defined in the Archer class. Instead, it should have been defined in the superclass. The loop tests that the private getters and setters that are common are not defined in the specific classes (Archer, Knight, and Orc).

6 | EMPIRICAL EVALUATION

The approach presented in this paper has been evaluated in a second-year programming university course, called Intermediate Programming, held at the University College Cork (UCC). This course mostly focuses on OOP (even if there are other contents as well). The assignment is done individually by each student. We do not mention the specific rubrics of the marks' distribution because, as mentioned, this course is composed of more content, as is often the case in university courses that teach OOP.

For evaluating the effectiveness of the AA approach, we selected two different groups, enrolled in the Intermediate Programming course. The first one was held in 2019 and the second one in 2020. All the students: (i) had no previous experience in OOP, (ii) were able to understand an object-oriented and (iii) were willing to answer a survey.

The students of 2019 completed the typical individual and nonincremental assignment in which wrapper classes are provided with the definitions of all the functions that the assignment requires (such as in the previous works [6] and [7]). It had a single submission (and consequently a single feedback) that was corrected manually. Whereas the students of 2020 completed the assignment presented in this paper (see Section 3) as a case study, which, as previously mentioned, is incremental and provides feedback twice with the AA.

Before doing this assignment, we observed that most of our students, even with good theoretical knowledge, still found it difficult and challenging the design and implementation of a solution for a real-world problem by using and applying the OOP principles. This changed for most of the students of 2020 (after developing the real case scenario assignment presented in this paper and getting its associated automatic feedback) while it was not the case for most of the students of 2019.

We would like to mention that, as previously mentioned, the assignment of the students of 2020 is presented in two parts. Then, the automatic feedback of the first part of the assignment should be provided as soon as possible, so that it can be used for the second part of the assignment. For the class evaluated in 2020, the students obtained on average more marks in the second part than in the first part of the assignment, which corroborates that the incremental particularity of the assignment (including its incremental feedback) worked well. This was not the case for the class of 2019.

Another point to consider is that it is also possible to combine both automated and manual assessments. Combining manual and automatic feedback means that both, automated and manual assessment can exist at the same time and support each other. For instance, if the design of the UML class diagram is a learning outcome of the course, it could be included as a part of the assignment and therefore the lecturer should manually mark it by comparing it with the one presented in this paper (see Figure 1).

Something that should be considered as well is that to clarify how the AA will be, some unit tests can be provided to the students. In this way, they can figure it out with more clarity the evaluation process. Especially, students not familiar with AA (and who might not trust AA yet). In addition, they will understand the importance in following exactly the assignment specifications. Furthermore, another advantage of this is that the students can learn better how unit testing works. Specifically, for the evaluation done for the 2020 class, a few samples of unit tests were provided to the students.

Another point to consider is how many re-submissions are allowed. Practice is important in learning programming and there should be room for mistakes and learning from them. AA can help as it can give feedback despite the limited human resources. However, to prevent mindless trial-and-error problemsolving, the number of resubmissions should be controlled [19]. In the evaluation of 2020, we asked the students to submit the first part of the assignment (Section 3.1), and then they get AA feedback. After a couple of weeks, they resubmit this part and the additional one (Section 3.2). However, the students of 2019 were only allowed to resubmit once (due to the lack of AA).

Following we present the results of the survey done to both classes about the assignment. Figure 3 shows the opinions of the students of 2019 (without the AA tool). We recall that this was a single nonincremental assignment. Forty-two students answered the survey.

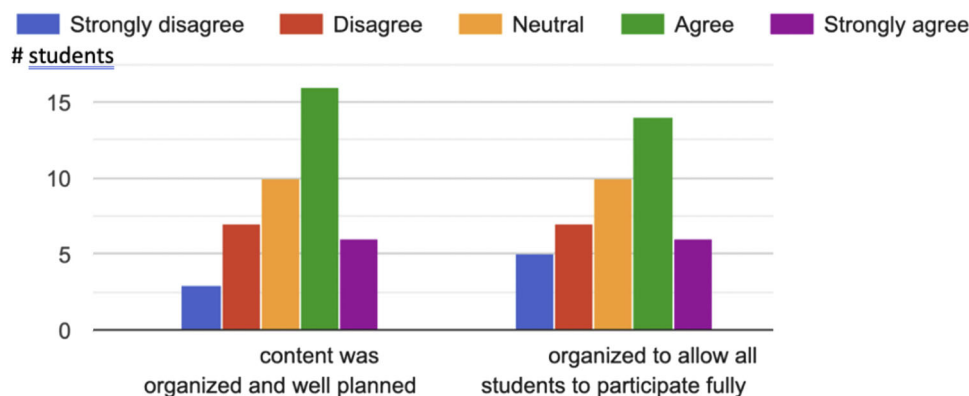


FIGURE 3 Results of the survey to the students of 2019 (without the AA tool).

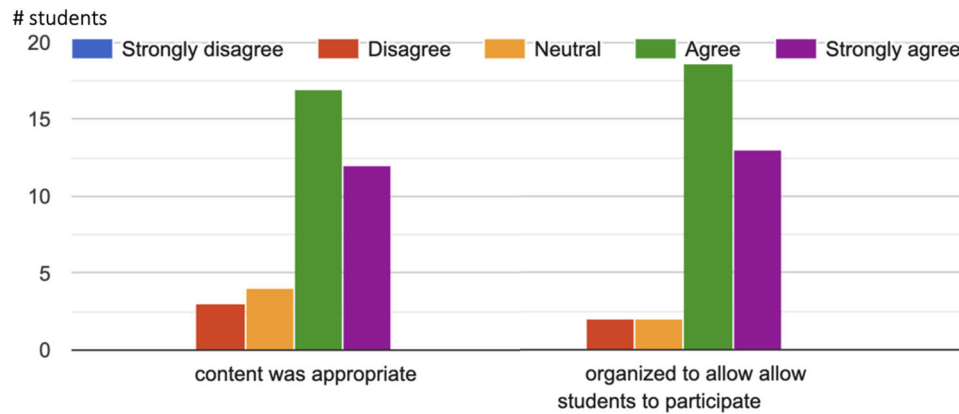


FIGURE 4 Results of the survey to the students of 2020 (with the AA tool).

It can be observed that 10 students (23.81%) had negative or strongly negative opinions about the content and almost the same (12 students, 28.57%) disagree/strongly disagree about the students' possibility of participation. Then, 10 students (23.81%) were neutral. Overall, this shows that around 50% of the class did not have a positive opinion.

The next year, in 2020, the new students experienced the AA tool and the real case assignment. After asking the class to complete a survey, 36 students answered. Figure 4 shows the results. The first change that can be observed with the surveys of the previous year (Figure 3) is that there are no students that have a strongly negative opinion. Only 2(5.56%)/3(8.33%) students had a negative opinion and the neutral opinions were reduced to 4(11.11%)/2 (5.56%) students. This year students had much more positive/strongly positive opinions (29/32 students, which is 80.56%/88.89%). It is even more noticeable the increase in the strongly agreed opinions since the previous year was 14.29% and the next year increased to one-third of the class. Therefore, the students valued the content and the possibility of participation with the new AA tool and incremental real case assignment.

Furthermore, the students who experienced the AA tool and real case assignment presented in this paper also wrote the following opinions when they were asked what aspects of this course were most useful or valuable: “using the concepts of OOP to slowly build on our knowledge of programming,” “the two assignments were very good for becoming proficient in object-oriented,” “the code examples and the two assignments helped me to learn the concepts taught in class,” “course content was interesting, assignments were a good way to learn and develop skills,” and so on (the survey contains more similar comments to these).

We believe that the positive scores and comments above-mentioned are due to several reasons. The students learn how to apply the OOP principles by

coding them. Because the students have almost complete freedom for designing the structure of the classes, rather than doing a monotonous assignment in which the students have to “refill” some given wrapper classes. In addition, the assignment is incremental, which motivates the students because they receive feedback from the first part before the start of the second part, allowing them to learn from their mistakes.

We would like to highlight that the case study and the AA presented in this paper allow the students' attainment of the difficult to asses LOs mentioned in Section 1.3. Specifically: (i) designing correctly the classes especially, in a project that grows over time, (ii) identifying the inheritance, aggregation, and composition relationships; and (iii) customizing the behavior of subclasses by overriding. Note that, as previously mentioned, these LOs can not be attained by the typical assignments, such as the assignment of the class 2019, the assignments of the previous works [6] and [7], and so on. Specifically, for the evaluation done for the 2020 class, a few samples of unit tests were provided to the students.

Furthermore, in terms of the correctness of the AA tool, a comparison of the results that are achieved by the AA approach presented, with the evaluation done by visual inspection of the student's code confirms the effectiveness of our designed approach.

7 | CONCLUSIONS AND FUTURE WORK

In this paper, we present an AA approach for OOP teaching and assessment (which, as mentioned in Section 1.1, it presents certain particularities and difficulties). Then, in this paper, we contribute to the literature by presenting an AA tool in Python that assesses complex OOP concepts (including class design)

such as inheritance, aggregation, overloading of operators, overriding, generalization/specification, error handling for protected attributes, and so on. In addition, it assesses the correctness of their implementation details. Furthermore, the unit testing approach presented in this paper also contributes to the students' incremental feedback; and the students learn unit testing and how to apply it to OOP programming problems.

Furthermore, we present a real case scenario of an assignment, for whom, the unit tests ensure that the code submitted by the students fulfills the specifications of such an assignment. The real case scenario assignment and the corresponding unit tests that we present, have the particularities and advantages that include all the OOP principles (Section 1.3) and incorporate them into a single incremental assignment divided into two stages.

In summary, the work presented in this paper is especially helpful for the students, since as previously mentioned, they typically struggle to apply the OOP principles, especially, if they have to apply all of them in a single and extensive programming assignment. The evaluation of our approach in a second-year course at a university, together with the feedback provided by the students, confirms the effectiveness of our approach. Therefore, positive experience has been gained in teaching OOP laboratory using the AA and the real-case scenario assignment presented in this paper.

The approach presented in this paper addresses the issues in teaching OOP Laboratory (some of them are present in other courses as well): (1) keeping the students motivated; (2) teach and assess not only the theoretical OOP (complex) principles but also how to practically apply them (in python); (3) incremental assignments and automatic feedback; and (4) the AA allows that a considerable low time is spent on corrections even for large groups. It is found that the current design of the OOP Laboratory presented in this paper adequately addresses all these issues.

As future work, we consider that the real case scenario could be extended (depending on the difficulty of the module, lab hours, etc.). Currently, there are three functionalities associated with the characters: the `__str__` method, the `>` operator, and the `fight` method. We could also add some new functionality. For example, some specific functionality to the Archer class. We also would like to develop different real case scenarios with the same particularities and advantages as the one presented in this paper, which is that it is incremental (the software requirements evolve with time) and combines all the OOP principles into a single assignment.

As future work, we also consider the creation of a tool that automatically creates the unit tests with automatic grading and learning feedback. There is previous work on this matter for OOP in the industry. But as far as we

are aware, there is no such tool for OOP for teaching. For example, evolutionary testing (ET) has been shown to be successful in automatically generating relevant unit test cases for procedural software [20].

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions that helped to improve the paper.

CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

Data sharing not applicable to this article as no data sets were generated or analyzed during the current study.

ORCID

Laura Climent  <http://orcid.org/0000-0001-9453-5150>

Alejandro Arbelaez  <https://orcid.org/0000-0003-1622-5645>

REFERENCES

1. W. K. Chen and Y. C. Cheng, *Teaching object-oriented programming laboratory with computer game programming*, IEEE Trans. Educ. **50** (2007), no. 3, 197–203.
2. D. Dang, *Teach all OOP principles in a single solution and expanding to solve similar problems*, CITREZ conference. 2007.
3. M. Kölling, *The problem of teaching object-oriented programming, part 1: languages*, J. Object-oriented Prog. **11** (1999), no. 8, 8–15.
4. N. Khamis and S. Idris, *Investigating current object oriented programming assessment method in Malaysia's universities*, Proceeding of ICEEI. 2007.
5. K. Srinath, *Python-the fastest growing programming language*, Int. Res. J. Eng. Technol. **4** (2017), no. 12, 354–357.
6. M. Torchiano and M. Morisio, *A fully automatic approach to the assessment of programming assignments*, Int. J. Eng. Educ. **25** (2009), no. 4, 814–829.
7. M. Torchiano and G. Bruno, *Integrating software engineering key practices into an oop massive in-classroom course: an experience report*, Proceedings of the 2nd international workshop on software engineering education for millennials. 2018, pp. 64–71.
8. P. Ihanntola, T. Ahoniemi, V. Karavirta, and O. Seppälä, *Review of recent systems for automatic assessment of programming assignments*, Proceedings of the 10th Koli calling international conference on computing education research. 2010, pp. 86–93.
9. E. G. Barriocanal, M. Á. S. Urbán, I. A. Cuevas, and P. D. Pérez, *An experience in integrating automated unit testing practices in an introductory programming course*, ACM SIGCSE Bull. **34** (2002), no. 4, 125–128.
10. J. L. Whalley and A. Philpott, *A unit testing approach to building novice programmers' skills and confidence*, Proceedings of the thirteenth Australasian computing education conference, vol. **114**, 2011, pp. 113–118.

11. S. Comb  fis and A. Paques, *Pythia reloaded: An intelligent unit testing-based code grader for education*, Proceedings of the 1st international workshop on code hunt workshop on educational software engineering, 2015, pp. 5–8.
12. F. Tour   and M. Badri, *Prioritizing unit testing effort using software metrics and machine learning classifiers (S)*, CITREZ conference. 2018, pp. 653–652.
13. S. Wappler and J. Wegener, *Evolutionary unit testing of object-oriented software using strongly-typed genetic programming*, Proceedings of the 8th annual conference on Genetic and evolutionary computation. 2006, pp. 1925–1932.
14. J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega, *Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software*, Inform. Software Technol. **51** (2009), no. 11, 1534–1548.
15. I. H. Hsiao, S. Sosnovsky, P. Brusilovsky, *Adaptive navigation support for parameterized questions in object-oriented programming*, European conference on technology enhanced learning. 2009, pp. 88–98.
16. K. M. Rajashekharaiyah, M. S. Patil, and G. Joshi, *Impact of classification of lab assignments and problem solving approach in object oriented programming lab course: a case study*, 2014 IEEE international conference on MOOC, innovation and technology in education (MITE) IEEE. 2014, pp. 205–209.
17. Y. C. Cheng, *Applying how to solve it in teaching object-oriented programming and engineering practices*, Aisan-PLoP. 2014.
18. C. Boudia, A. Bengueddach, H. Haffaf, *Collaborative strategy for teaching and learning object-oriented programming course: a case study at mostafa stambouli mascara university, Algeria*. Inform. **43** (2019), no. 1. <https://doi.org/10.31449/inf.v43i1.2335>
19. L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, *Experiences on automatically assessed algorithm simulation exercises with different resubmission policies*, J. Educ. Resour. Comput. **5** (2005), no. 3, 7–es.
20. P. McMinn, *Search-based software test data generation: a survey*, Softw. Test. Verif. Reliab. **14** (2004), no. 2, 105–156.

How to cite this article: L. Climent and A. Arbelaez, *Automatic assessment of object oriented programming assignments with unit testing in Python and a real case assignment*, Comput. Appl. Eng. Educ. (2023), 1–18.
<https://doi.org/10.1002/cae.22642>

APPENDIX A: ADDITIONAL CODE

Here we present the additional code mentioned in Section 5. Specifically, the function that captures the output (`capt_out()`), its imports and the rest of the unit tests.

A.1. Function for capturing the output

First, we specify the imports associated with the function for capturing the output and subsequently we define the function.

```
from contextlib import contextmanager
try: # Python 2
    from StringIO import StringIO
except ImportError: # Python 3
    from io import StringIO

@contextmanager
def capt_out():
    new_o, new_e = StringIO(), StringIO()
    old_o, old_e = sys.stdout, sys.stderr
    try:
        sys.stdout, sys.stderr = new_o, new_e
        yield sys.stdout, sys.stderr
    finally:
        sys.stdout = old_o
        sys.stderr = old_e
```

A.2. Constructors

Below we present two extra unit tests for checking the types of values and the range of the values for archers and knights.

```
def test_values_types_constr_archer(self):
    with capt_out() as (out, err):
        mod.Archer("archer1", 8)
        output = out.getvalue().strip()
        assert (output == "type ERROR")

def test_values_types_constr_knight(self):
    with capt_out() as (out, err):
        mod.Knight("k1", 3.0,
                    "Gondor", ["AAB"])
        output = out.getvalue().strip()
        assert (output == "archers list ERROR")
        a1 = mod.Archer("a1", 3.3, "Atla")
        a2 = mod.Archer("a2", 2.3, "Mordor")
        k2 = mod.Knight("k2", 4.0, "Atla",
                        [a1, a2])
        assert k2.archers_list == [a1]
```

A.3. Properties

Here we include unit tests for checking the access and update of the properties of the orcs, including the incorrect uses of the properties.

```
def test_properties_access_orc(self):
    orc = mod.Orc("Ogrorg", 4.1, True)
    assert orc.name == "Ogrorg"
    assert orc.strength == 4.1
    assert orc.weapon
```



```
orc.name = "Grunch"
assert orc.name == "Grunch"
orc.strength = 3.2
assert orc.strength == 3.2
orc.weapon = False
assert not orc.weapon
```

```
def test_properties_values_errors_orc
(self):
    orc = mod.Orc("Ogrorg", 4.1, True)
    with capt_out() as (out, err):
        orc.name = 1.2
    output = out.getvalue().strip()
    assert (output == "type ERROR")
    with capt_out() as (out, err):
        orc.strength = "Grunch"
    output = out.getvalue().strip()
    assert output == "type ERROR"
    with capt_out() as (out, err):
        orc.weapon = 6.8
    output = out.getvalue().strip()
    assert (output == "type ERROR")
    orc.strength = 10.0
    assert orc.strength == 5.0
    orc.strength = -10.0
    assert orc.strength == 0.0
```

A.4. `__str__` method

Below, we present the two unit tests of the `__str__` method for orcs and knights.

```
def test__str__orc(self):
    orc = mod.Orc("Ogrorg", 3.3, True)
    with capt_out() as (out, err):
        print(orc)
    output = out.getvalue().strip()
    assert (output == "Ogrorg 3.3 True")

def test__str__knight(self):
    a1 = mod.Archer("a1", 3.3, "At1")
    k1 = mod.Knight("k1", 4.0, "At1", [a1])
    with capt_out() as (out, err):
        print(k1)
    output = out.getvalue().strip()
    assert (output == "k1 4.0 At1
[a1 3.3 At1]")
```

A.5. `>` operator

We present an extra unit test for the `>` operator used by different orcs with and without a weapon.

```
def test_greater_orc(self):
    orc1 = mod.Orc("o1", 3.3, True)
    orc2 = mod.Orc("o2", 4.9, False)
    assert orc1 > orc2
    orc3 = mod.Orc("o3", 3.3, False)
    assert not orc3 > orc2
```

A.6. Fight method

The last extra unit test checks that the limits of the strength of the characters are correctly truncated after fighting.

```
def test_fight_truncation_orc(self):
    orc1 = mod.Orc("Ogrorg", 3.3, True)
    orc2 = mod.Orc("Grunch", 4.9, False)
    orc1.fight(orc2)
    assert (orc2.strength == 4.9 and
            orc1.strength == 4.3)
    orc1 = mod.Orc("Ogrorg", 3.3, False)
    orc1.fight(orc2)
    assert (orc2.strength == 5.0 and
            orc1.strength == 3.3)
    orc1 = mod.Orc("Ogrorg", 0.1, False)
    orc2 = mod.Orc("Grunch", 0.1, False)
    orc1.fight(orc2)
    assert (orc1.strength == 0 and
            orc2.strength == 0)
```

AUTHOR BIOGRAPHIES



Laura Climent's obtained her PhD in Computer Science from the Universitat Politècnica de València (UPV). The research of her thesis is within the area of Combinatorial Optimisation. After finishing her PhD, she worked as a post-doctoral researcher at the University College Cork, UCC (2013–2016). She was also a lecturer in the School of Computing and Information Technology at UCC. Currently, she is a lecturer in the Computer Science Department in the Universidad Autónoma de Madrid (UAM). Dr. Climent has participated in several Spanish and Irish research projects. As a result, she has authored more than 30 peer-reviewed publications in conferences and journals: 7 journals ranked in JCR and SCOPUS and more than 20 conference papers. Among the journals, it can be highlighted the publication in the *Journal of Artificial Intelligence Research (JAIR)*, a well-known and prestigious journal in the Artificial Intelligence field. In addition, she has

been a reviewer of several international conferences and journals. She also served as a program chair of the doctoral program of the prestigious CP conference and she supervised a PhD student in UCC.



Alejandro Arbelaez received his PhD in computer science from University of Paris XI for his work on applying learning-based techniques to solve combinatorial problems. After finishing his PhD, he worked as a postdoctoral

researcher at the University of Tokyo (2011–2013) and University College Cork, UCC (2013–2016). He was also a lecturer in the School of Computing and Information Technology at UCC. Currently, he is a lecturer in the Computer Science Department in the Universidad Autónoma de Madrid (UAM). Dr. Arbelaez is the author of more than 40 research papers in prestigious conferences, journals, international workshops, and two chapters in prestigious books of the Artificial Intelligence field.