Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3341525.3387369

RESEARCH-ARTICLE

Postponing the Concept of Class When Introducing OOP

NICOLÁS PASSERINI, National University of San Martín, San Martin, Provincia de Buenos Aires, Argentina

CARLOS LOMBARDI, National University of Hurlingham, Hurlingham, Provincia de Buenos Aires, Argentina

Open Access Support provided by:

National University of San Martín

National University of Hurlingham

PDF Download
3341525.3387369.pdf
01 January 2026
Total Citations: 1
Total Downloads: 264

Published: 15 June 2020

Citation in BibTeX format

ITiCSE '20: Innovation and Technology in Computer Science Education
June 15 - 19, 2020
Trondheim, Norway

Conference Sponsors:
SIGCSE

https://doi.org/10.1145/3341525.3387369
ISBN: 9781450368742

Postponing the Concept of Class When Introducing OOP

Nicolás Passerini
Universidad Nacional de Quilmes
Bernal, Buenos Aires, Argentina
Universidad Nacional de San Martín
San Martín, Argentina
npasserini@gmail.com

Carlos Lombardi
Universidad Nacional de Hurlingham
Villa Tesei, Buenos Aires, Argentina
Universidad Nacional de Quilmes
Bernal, Buenos Aires, Argentina
carlombardi@gmail.com

ABSTRACT

The literature on programming education describes different problems found in courses that introduce the basic concepts of Object-Oriented Programming (OOP). Some of these problems arise from the large amounts of abstract concepts that are needed even for the simplest programs. Other difficulties are related with the concepts of class and instantiation, and the duality between classes and objects. Educators and researchers have proposed several alternatives to define a gradual path for the introduction of OOP.

A group of educators from several universities in the Buenos Aires area crafted a learning path for a first course about OOP in which the concepts of class and instantiation are introduced several weeks after the beginning of the course. Gradualism is achieved in this proposal by starting with a minimal metamodel based on self-defined objects, which is progressively enlarged. Following this learning path, by the time students are introduced to classes and instantiation, they already have a good acquaintance with object definition and interaction, and are also able to quickly understand the convenience of the new concepts.

The same group conceived and developed a didactically-oriented programming language along with an IDE; and produced several exercises that can be solved using the initial metamodels.

In this article, we discuss which concepts and language elements can be introduced before classes and instantiation, the need for a programming language that supports the proposed learning path, and the results of its application in several universities.

ACM Reference Format:
Nicolás Passerini and Carlos Lombardi. 2020. Postponing the Concept of Class When Introducing OOP. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–19, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341525.3387369

1 INTRODUCTION

Learning to program is, in general, a hard task for many students. This fact has been widely reported in literature [4, 13, 23, 32, 33], and testified by the high dropout rates in first-year courses [7, 16]. Several well-documented difficulties hinder novice students ability to grasp the basic programming concepts and techniques. On the other hand, the strong influence of Object-Oriented Programming (OOP) in the state of the art of professional software development, yields the relevance of integrating this programming paradigm in university programming-related curricula.

Literature about computer science education includes many works about initial OOP courses. Some of the obstacles described in these works are common to initial programming courses, as the high amount of concepts and details that are needed in some languages or programming environments to produce even the simplest running program[26], or the inadequacy of professional environments for the use by novice programmers[24]. Among the hindrances specific to OOP, several works underline those derived from the duality between classes and their instances and the instantiation operation [12, 22, 23, 30, 35], that provoke confusion in many students. Other authors, in particular Guzdial [20], state that previous experience in other programming paradigms could lead to a *centralized mindset* that undermines the possibility of making best use of critical OOP tools such as collaboration and polymorphism.

A first course on OOP deals with a basic theory that involves several abstract concepts, and also with the syntactic intricacies of a programming language, hence the need for a *gradual* approach to OOP. The literature includes different proposals regarding this goal [4, 10, 18].

We want to remark that, while several authors underline the relevance of prioritizing, in OOP courses, basic concepts over the details of a particular programming language or technology[4, 15, 28], most works focus on the description of particular tools, with little or no relation with broader didactical ideas.

During the last 15 years, a group of teachers from several universities in the Buenos Aires metropolitan area have elaborated a pedagogical proposal for OOP initial teaching. A key aspect of this proposal is that the concepts of class and instantiation are introduced after several weeks. Up to that point, students work with self-defined objects, or *well-known objects* (WKO) that involve the main elements of a class (i.e. attributes and methods), and have a globally-accessible name; WKOs are singletons that do not need to be instantiated. This initial setting allows to introduce several of the main themes in OOP: object, state, references, polymorphism and parameters.

This approach also raises the need for a language that allows for the definition of self-defined (*i.e.,* class-less) objects. The examples in this paper, as well as the latest experience implementing these ideas in real classrooms are based on an educative language named Wollok; yet other tools could and have been used with the same objective, e.g. Scala or Javascript among other industrial languages.

Wollok is an educative language as well as an ecosystem of educative programming tools, both of them conceived specifically

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ITiCSE '20, June 15–19, 2020, Trondheim, Norway
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6874-2/20/06…$15.00
https://doi.org/10.1145/3341525.3387369

for the implementation of these and other related ideas. Unlike Javascript, the same syntax is used to define classes or WKOs, which allows for a really soft transition to a latter stage of the course in which classes and instances are introduced.

In a previous work [29], we focus on the Wollok language and programming environment. The main contribution of this paper is a detailed description of an approach for introducing OOP in first year university courses. One disctinctive characteristic of this approach is a gradual path for introducing OOP concepts, starting with a minimal metamodel based on self-defined objects, which is progressively enlarged, allowing the students to grasp the concepts one by one. Introduction of classes and instantiation is postponed to a later stage, which allows to avoid several common pitfalls in the first part of a course. This approach is supported with many examples carefully shaped to be solvable without the need of classes.

The application of the learning path described here goes back to 2006, although several parts of it have evolved with the years. In these 14 years, our approach, while evolving, has been applied by more than 100 teachers and 7000 students, both at univeristary and highschool levels. The accumulated experience revealed significant improvements in approval rates, student retention and understanding of OOP concepts. In particular, the approach favors that students naturally apply polymorphism in their designs from a very early point in the course, allowing for more practice on polymorphic object models during the course and, thus, more complex design patterns in the last part of it.

Our work is mainly focused on first year university courses and most frequently applied on universities with a considerable population of students coming from families with limited income. For example, many students are first generation of university students and usually are still in the process of adapting to university rules and requirements; these conditions increase the probability of dropouts in the first year of the career [17]. Therefore, one of our main goals is the retention of vulnerable students, as part of an inclusion policy that considers university education as an essential tool for social mobility and the reduction of social inequality [11].

Paper outline. In Sec. 2 we discuss the literature related with initial OOP teaching. In Sec. 3, we describe how many fundamental concepts of OOP can be introduced without using classes and instantiation, and features of Wollok that give support for each. In Sec. 4 we point out some aspects about course organization and educational tools. Sec. 5 details several indicators about the accumulated experience applying the ideas described in this article, and Sec. 6 includes some conclusions and ideas for future development.

2 RELATED WORK

The literature about computer science education include many articles related with the introductory teaching of OOP. On the other hand, to the best of author's knowledge, only few of these works combines a focus on the description of a comprehensive pedagogical proposal with the introduction of languages and/or educational tools motivated in terms of their adequacy with the pedagogical ideas. To this respect, we point out the works of Meyer [28] and Kölling et al. [24]. Both works share a strong interest about proposing a gradual approach, and also the strategy to achieve this aim, based on the nature of the tasks students are faced to. At the beginning, activities involve just using definitions provided by the course. Subsequently, students modify existing methods and/or add methods to existing classes; then they asked to add classes to a given design, and finally, to produce their own solutions to a given statement. This approach conforms to the "Fill in the Blanks" educational pattern described by Bergin [6].

Different approaches to graduality can be found in previous works. Bennedsen [4] focuses in "graduated exposure to complexity and structure": students are presented very simple models in the beginning, and their complexity is gradually increased through the course. This work also proposes to provide students with a design, described through UML diagrams, so that they must build an implementation. In [18], Gray and Flatt describe a series of successive "language levels", that include a carefully designed subset of the Java syntax, and also define enforced restrictions (e.g. in the beginner level `if` statements must have `else`), in order to drive student attention to the specific material discussed at each part of the course. Interestingly, inheritance is included in the first level, while arrays are postponed to the third one. We also mention that Cazzola and Olivares [10] describe a method to gradually teach a programming language. This work suggests to define a relationship between programming concepts and syntax elements, and to derive a series of sublanguages from that definition.

Several educational programming environments appear in the literature as well. Among those, BlueJ [24], has been described as "the most popular educational tool" in [15], see also [1, 5, 8, 21, 25, 30, 31, 36]. BlueJ is an environment for Java that designed to ease coding, program visualization and exploration. In particular, it lets students to interact with objects in a simple way through a GUI that allows to create objects from the available classes, and subsequently invoke methods on those objects. It also includes facilities for debugging and automated testing, beyond program edition, evaluation and visualization. Many other educational programming environments, most of them based on industrial languages, have been proposed; we mention ProfessorJ [18] and DrJava [2] for Java, and Thonny [3] for Python.

The proposal for an initial OOP course we describe in this article integrates many elements from the above described works. One is the focus on model building promoted by Bennedsen [4] and Meyer [28]: our courses are organized by means of a series of problem domains of increasing complexity[1], and each major concept is motivated through a requirement in the context of a domain. We also combine the dynamic and static perspectives on an OOP program as described in [28], through the extensive use of object diagrams fautomatically generated by the Wollok IDE. Additionally, conforming to the "Fill in the blanks" educational pattern [6, 24], some domains used in the courses are partially implemented, so that students can immediately interact with the implemented objects or classes, and also practice the combination of their own code with a preexistent code base. Finally, we mention that we share with [24] the aim of allow interaction with objects without the exposition of the several concepts involved in Java `main` construct.

[1] In our courses, problem domains range from a streamlined model of a bird and its food (see Sections 3.1 and 3.2) to models that involve the interaction of many objects, including some pertaining to different class hierarchies, to solve a specific requirement.

On the other hand, we note that all the works mentioned in this section operate with classes and instantiation right from the start (or in any case, in the stage in which OOP is introduced). The possibility of postponing these concepts via an initial setting based in objects that are not instances of a class, with the analysis about the extent in which such idea can be applied and its impact on the early exposure to other concepts like polymorphism or references, is not mentioned in any work known by the authors.

We mention another distinct aspect of our proposal, namely the criterion taken to organize the gradual exposure to OOP. In turn of language features or model complexity, we gradually introduce, and work with, OOP constructs, from just objects, their definition and relationships, to collections, then classes, and finally class inheritance.

3 LIFE WITHOUT CLASSES

In this section, we describe a way to introduce several of the main concepts about OOP, in a setting that does not resort to classes or instantiation. The material in this section is based in the accumulated experience of the authors, along with several colleagues. It fits for a second-semester, first-year course, that has as prerequisite an initial course on programming. Along with each concept, we describe elements of the Wollok language and IDE that support the proposed approach, and examples that we actually use.

3.1 Object definition

We start by describing an object as a computational model of an entity that exhibits some behavior, in the form of messages that the object can be asked.

A typical initial example is a model of a bird, whose energy change patterns are to be coded. It involves four operations: to fly (that decreases the available energy), to eat (that increases it), to give the current amount of energy, and to tell whether it is hungry, which is true if the energy is below 50. In turn, in order to behave as expected, the bird model must *memorize* the amount of energy; this observation leads to the introduction of the object state.

Wollok allows very simple definitions of well-known objects.

```
object pepita {
  var energy = 0
  method energy() { return energy }
  method fly(kms) { energy -= kms + 10 }
  method eat(grams) { energy += grams * 4 }
  method isHungry() { return energy < 50 }
}
```

The Wollok IDE includes a REPL (Read-Evaluate-Print Loop) console, that allows to quickly interact with the defined objects.

```
Wollok REPL Console: pepinha/src/pepita.wlk
Consola Interactiva de Wollok (escriba "quit" para salir):
>>> pepita.isHungry()
true
>>> pepita.eat(100)
>>> pepita.fly(150)
>>> pepita.isHungry()
false
>>>
```

3.2 Interacting objects, polymorphism

As soon as students get familiar with the syntax and the IDE, we move on to a slightly complex setting: the `eat` operation must be supplied with the food `pepita` is actually eating, along with the grams. The amount of energy in each food gram, fixed to 4 in the previous setting, now depends on the eaten food. The `eat` method changes to the following:

```
method eat(food, grams) {
  energy += grams * food.energyPerGram()
}
```

Among the possible foods we propose: canary grass, having 4 energy units/gram, and millext, whose energy depends on whether it is wet (10) or not (15 units/gram). The need for objects to which `energyPerGram()` can be asked to, is quickly sensed by students.

```
object canaryGrass {
  method energyPerGram() { return 4 }
}
object millet {
  var isWet = false
  method becomeWet() { isWet = true }
  method becomeDry() { isWet = false }
  method energyPerGram() {
    return if (isWet) { 10 } else { 15 }
  }
}
```

Based on an initial example, students immediately code the `canaryGrass` object. While the `millet` imposes the problem of understanding the need for an object state, the need of an `energyPerGram()` method is perceived by most students.

Following these lines, *polymorphism* becomes intuitive for most students: they just realize that different objects (the food) need to respect a common contract (`energyPerGram()`) so that any of the can be used by a third party (the bird). Situations where polymorphism needs to be exploited appear several times in our courses; this concept does not trouble students in most cases. See Section 5.2.

3.3 References

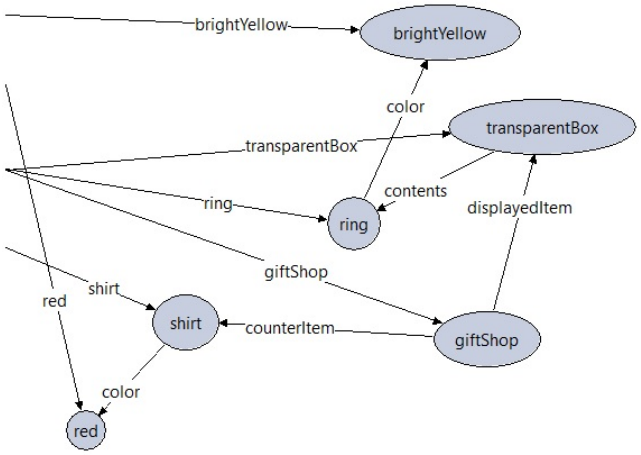
WKOs suffice to introduce, and work with, references among objects. One of the statements we designed to this end is a tiny model of a gift shop that includes objects of several kinds: colors, gifts, the gift shop, and people. One of the gifts is a transparent box, that can hold any of the other gifts. In turn, the gift shop stores two items: one is displayed, the other one is in the counter. An excerpt from the implementation follows. The complete domain includes more colors (including `brightYellow`), gifts (including a `ring` and a `shirt`) and persons, many more methods in the `giftShop` object, and several getter and setter methods.

```
// a color
object red { method shines() { return true } }

object transparentBox {      // a gift
  var contents
  method setContents(cnts) { contents = cnts }
  method color() { return contents.color() }
  method weight() { return contents.weight() + 200 }
}

object giftShop {            // the shop
  var displayedItem ; var counterItem
  method isBrilliant() {
    return displayedItem.color().shines() and
           counterItem.color().shines()
  }
}
```

```
object laura {        // a person
  method likes(gift) { return gift.weight() < 500 }
}
```

This simple domain allows to note how behavior depends deeply on dynamic relationships and interaction among many objects. In this case, several objects participate to build up the answers for `laura.likes(transparentBox)`, and even more for `giftShop.isBrilliant()`.

The Wollok IDE includes automatically generated object diagrams, that display the state comprising the defined objects and their connections. An example for the described domain follows.



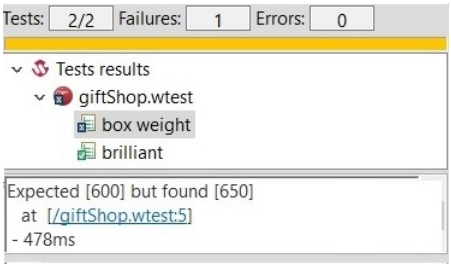
The object diagram has proven to be a useful device to help students understand how objects are related, and the contribution of many objects when a certain operation is requested.

3.4 Introduction to automated testing

The first experiences on automated testing usually precede, in our courses, the introduction of classes. Since each WKO has a globally accesible name, it is easy to produce tests about them.

The Wollok language allows test definitions[2]; in turn, a test runner is integrated in the IDE. We show two tests about the gift shop domain, and the output of the test runner for them.

```
test "box weight" {
  transparentBox.setContents(ring)
  assert.equals(600, transparentBox.weight())
}
test "brilliant" {
  transparentBox.setContents(ring)
  giftShop.setItems(transparentBox, shirt)
  assert.that(giftShop.isBrilliant())
}
```



[2]We note that test syntax is very simple, and yet similar to that of industrial libraries, e.g. Jest for Javascript.

Following good testing practices, WKOs are re-created before each test. This feature is crucial to preserve test independence, since students cannot control object lifecycle.

3.5 Collections and first-class functions

The Wollok language includes literals to build lists and sets, as well as anonymous functions that are used in queries on collections, as shown in this REPL:

```
Wollok REPL Console: pepinha/src/giftShop.wlk
Wollok interactive console (type "quit" to quit):
>>> [ring,shirt,ball].filter({gift => gift.color().shines()})
[ring[color=brightYellow[]], shirt[]]
>>> [ring,shirt,ball].map({gift => gift.weight()})
[450, 800, 600]
>>>
```

These syntactic devices allow to work with collections with focus on their behavior and the nature of each query (whether we want to filter based on a condition, to map through a transformation, to test whether all/any of the elements hold a condition, etc.), avoiding both algorithmic details that detour the focus from the result expected from each operation, and the discussion about the class each collection belongs to.

3.6 Transition to classes

Several problem domains used in the initial stages admit natural extensions that motivate the introduction of classes. E.g. in the gift shop domain, we can pose the following issue: how to extend the model to admit several transparent boxes, each holding a different gift, many shirts of different colors, or more than one gift shop.

The issue is intuitive for the given domain; in fact, these kind of questions are often posed by students. Moreover, sometimes a student come with the perspicious remark: "there should be some way to *create new* transparent boxes". This situation makes an excellent context for the introduction of classes and instantiation.

The syntax of the Wollok language greatly simplify the initial experiments with classes, since class definition is nearly identical to that of WKOs students are already accustomed to at this point. To transform a WKO definition into a class definition consists in just changing the word `object` by `class`, and capitalize the name:

```
class TransparentBox {
  var contents
  method color() { return contents.color() }
  method weight() { return contents.weight() + 200 }
}
```

Instantiation is applied by a standard use of the `new` keyword.

```
const transparentBox = new TransparentBox()
```

4 DISCUSSION

Section 3 shows the strategy we propose to fulfill the graduality requirement: by working with a series of growing *object metamodels*, i.e. descriptions of OOP from which the model of each presented domain is built upon.

We organise our courses in four stages, each corresponding to a metamodel. The first stage involves just WKOs as model building blocks. As described in Secs. 3.1 to 3.4, WKOs suffice to include references, polymorphism and the dynamic vision; and even introduce automatic testing. The second stage integrates collections

and first-class functions, *cf.* Section 3.5. In this stage, the WKOs can manage collections of related objects; e.g., the gift shop can now handle many items in the counter. The third stage adds classes and instantiation, it is motivated as we describe in Section 3.6. The fourth stage involves inheritance.

In turn, each metamodel correlates with a subset of the Wollok language. It is important to note that the language was designed specifically to go with the course organisation, to the contrary of course proposals based on the details of a particular language; *cf.* the critics to the "syntax-driven organization of the material" in [4].

We put special care about the *transition* between stages: each extension of the metamodel is motivated by proposing an extension from an already known domain, such that the extension does not admit a fit solution with the tools available at the moment. In this way, the new concept (collections, classes, inheritance) is introduced when students had acquired fluency with a metamodel, and also feel the need to expand it in order to cope with a concrete situation. They are able to comprehend the new material, using it as the missing piece that let *them* solve the posed problem. Furthermore, all the concepts learned previously are perfectly valid in the new setting; in general, we care not to have students learn concepts that they need to *un-learn* later on. The example given in Section 3.6 about how classes are introduced testifies our approach on transitions.

The same example shows the interplay between the static and dynamic perspectives: the need for the existence of different objects that share their state and behavior definition is made clear through the use of the object diagram. In the gift shop example, if we want the shop items to be two shirts, one red and one blue, a possible option is to assign the `shirt` object as both items:

```
giftShop.setItems(shirt, shirt)
```

The two references from the `giftShop` object to the `shirt` object are evident in the object diagram. Some computations, e.g. the total weight of the items in the shop, work as expected. But now, if we want to change the color of the displayed item

```
giftShop.displayedItem().color(blue)
```

then the object diagram allows to note that the object whose color changed is, in fact, the *only* `shirt` object, so that if we ask

```
giftShop.counterItem().color()
```

we discover that it has "also" changed to `blue`. The need of having different objects to model each shirt becomes apparent.

The organisation in stages allows us to follow a *spiral* pattern [4, 6, 34] regarding several key concepts, as polymorphism or the dynamic perspective. Take e.g. the latter. In the initial stage, object diagrams include just simple WKOs and their references. The second stage involves lists and sets, leading to more complex diagrams. In turn, the third stage adds objects that are created dynamically, and that have no "given name" (e.g. if we set the items in the gift shop like this: `giftShop.setItems(new Shirt(), new Ring())`); the gradual progression of metamodels through the course greatly helps students to understand the many variants present.

Finally, we want to emphasize the relevance of using a programming language that favors student exploration and comprehension of OOP principles and notions. One of the design aims of the Wollok language is to allow WKO definitions with a minimum of syntax. This decision, along with the REPL and other IDE features (as syntax coloring, omission of a separate compilation phase, and error messages tailored to initial students), makes it easy for students to experiment right from the first session. Conversely, *cf.* the counter-intuitive nature of some Java syntactic constructs noted in [9].

On the other hand, literature signals several weak points of the use of educational languages; among the most cited we find the low value assigned by students, more interested in learning tools used in industry, and the deficient transfer of the acquired knowledge in subsequent courses or professional activity in which industrial languages and IDE are used. We addressed the first criticism by including a question about the perceived value of the Wollok language in a survey answered by students of different universities. Regarding the second one, we point out the performance of students in one university in which a software design course that uses Java as programming language follows immediately the introductory OOP course. *Cf.* Section 5 for the details.

5 RESULTS

The application of a learning path that delays the use of classes goes back to 2006. In these 13 years, our approach, while evolving, has been applied by more than 100 teachers and 7000 students.

A relevant element of this evolution is the design, construction and use of pedagogical tools that go in line with the proposed approach, *cf.* [14, 19, 27]. Wollok represent the current stage in a line of work. In particular, since we started using Wollok in 2015, this tool has been applied in more than 60 courses in six different universities, reaching more than 2500 students. It has also been used successfully at highschool level.

In order to assess the results of the teaching proposal described in this article, we considered indicators that point to three different aspects. The pass rate of courses that follow these ideas in several universities let us check the degree of academic success. We include also data about a subsequent course from one university. In turn, a detailed analysis on the results of an assignment yields indicators about the ability of students to apply the taught concepts. Finally, a survey performed on students on the end of the course, allows to assess their perception about the learning process and its results.

5.1 Pass rates

We studied the performance of the courses in three of the universities that implement this approach in the first course about OOP: Universidad Nacional de San Martín (UNSaM), Universidad Nacional de Quilmes (UNQ) and Universidad Nacional de Hurlingham (UNaHur). The course belongs to the second (UNQ, UNaHur) or third (UNSaM) semester of a bachelor curriculum focused on programming. Students that quit the courses in the first 5 weeks, or quit them due to personal reasons [3] were excluded from the study.

We gathered student performance from 2015 to 2018; covering 15 courses with 327 enrolled students. A total of 247 students passed the subject, implying a 75.5% pass rate. If we consider only the students who took at least one of the two main evaluations in each course, the universe size is reduced to 296 students; the pass rate raises to 83.4%. Unfortunately, we can make no direct performance

[3] Many university students in Argentina work for a living, which makes course withdrawal a common situation.

comparisons, since there were no, or little, courses of the same subject before this approach was implemented in the target universities, or else the data is not available. Nonetheless, we remark that the obtained pass rates contrast with the consistent reports about low scores for initial programming courses. To cite a recent example, [7] describes an introductory programming course with 377 enrolled students, where a 20% pass rate over enrolled, or 32% considering only students having taken the final exam, is reported before specific actions to increase these figures were taken.

We also mention the perfomance of UNaHur students in a course about object-oriented software design, that uses Java as programming language, that follows immediately the course about OOP. This course was given twice in 2019, for a total of 50 enrolled students; 42 of them passed the subject, implying a 84% pass rate. We take this rate as an indicator that students are actually able to apply, in later courses, the knowledge obtained in the initial OOP course.

5.2 Detailed analysis of an exercise

We studied the solutions produced by students in a UNQ course, for an exercise given to them after 6 weeks of course (for a total of 48 hours that combine lectures and lab sessions), soon after classes were introduced. The domain is a simplified model of taxi agencies, organized in two evolutive parts. The first part involves four entity types, with polymorphic alternatives for three of them. E.g. there are four kinds of cars, which must show polymorphic behavior. In turn, some of the entity types are set out as singletons, so that they should be better modeled as WKOs rather than classes. Some items involve queries on collections that take a function as argument, namely `map`, `filter` and others. We added the specification for an automated test that requires to create 11 objects and manipulate 3 WKOs. The second part involves the addition of a new entity type, related to two entity types previously modeled.

A total of 28 students delivered the exercise in time, in a course with 30 enrolled students [4]. About the first part, 27 students could build a sound model; in particular, 23 could distinguish in a right way WKOs from classes. All the 28 students could build the scenario of the mandatory test involving the creation of the specified objects.

We found no problems about the use of polymorphism: the issues that required a polymorphic use of objects were successfully solved by all students, and no code testing for the kind of an object was present. In turn, we found only four cases in which some object is sent a wrong message: two are about using a collection as it were an element, one uses a model object as it were a number, the other one is a confusion between model objects. Regarding the use of collections, only 5 students showed problems, mostly with the use of `map`. Finally, regarding the second part, 22 students extended the model in a sound way, and 18 including some degree of testing, even when it was not explicitly asked in the statement.

These results reveal that first-year students, after 6 weeks, are able to: build a domain model using both classes and WKO, apply polymorphism in a right way, use collections combined with first-class functions, build an automated test that involves the creation of several objects, and augment a previously defined object model.

[4]The statement (in Spanish) is available in `https://github.com/wollok/remiseria`. Answers are not made public for student privacy reasons, ask the authors about them.

5.3 Perception of students

At the end of the (southern) spring 2018 and fall 2019 semesters, we performed a survey on a random selection of students pertaining to several courses that applied the ideas described in this article, with 150 answers. The surveys include several Likert questions with five options, some about like/dislike a proposed assertion, others about the perceived difficulty of some task or concept.

The most pertinent question regarding the subject of this article is "How easy/difficult was to you to work with instances when we switched from WKOs to classes?". We obtained 85% of positive ("Very accessible" or "accessible") answers.

Another important concern for us is our choice of an educative language, because of the recurrent pressure to use the most *trending* languages and tools. We decided to include a question about how the students perceive the applicability of the concepts they learned in the course, with 78% of positive answers.

These results suggest a consistently positive perception.

6 CONCLUSIONS AND FUTURE WORK

This article describes a strategy to introduce OOP that uses initially a minimal metamodel based in self-defined objects. The concepts of class and instantiation are incorporated in a later stage, when students have already acquired fluency in defining, using and relating objects. This strategy has been applied for more than 10 years in several universities in the Buenos Aires area. We observe that it contributes to the understanding, by a large proportion of students, of the concepts that form an initial OOP metamodel including object polymorphism; and to their ability to use these concepts in problem solving. Such observation can be testified by high pass rates, sound application of OOP concepts in lab exercises, and student perception. In our opinion, these results are especially relevant given the traits of the student population in several of the universities where this strategy is used.

Some features of the Wollok IDE, not discussed in this article, also contribute to achieve a positive student experience. We outline the early indication of errors in the code editor, in line with the practice of modern industrial tools, but where the error messages are tailored for freshman students. See [29].

The future work we envisage is related to the enhancement of this pedagogical strategy. Possible work directions include: a new version of the Wollok IDE available from a Web browser, to promote the adoption of both Wollok and the ideas described in this article; a revised version of lab activities that can be worked out with a OOP metamodel based on WKOs; and better evidence about the acquisition of programming abilities by students and also about their perceptions. In other respects, we also plan to gather information about how OOP is introduced in other programming-related curricula, in order to consider possible comparative studies.

ACKNOWLEDGMENTS

The authors would like to thank the team behind the development of Wollok and its related tools, as well as all the teachers that use it in their classes and have provided great feedback to refine the ideas presented in this work. This work has been partially supported by the Universidad Nacional de Quilmes under grants PUNQ 1327/19 and PUNQ 1439/15.

REFERENCES

[1] B. Alkazemi and G. Grami. Utilizing bluej to teach polymorphism in an advanced object-oriented programming course. *Journal of Information Technology Education: Innovations in Practice*, 11, 09 2012.

[2] E. Allen, R. Cartwright, and B. Stoler. Drjava: A lightweight pedagogic environment for Java. In *ACM SIGCSE Bulletin*, volume 34, pages 137–141. ACM, 2002.

[3] A. Annamaa. Thonny,: A python ide for learning programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 343–343, New York, NY, USA, 2015. ACM.

[4] J. Bennedsen and M. E. Caspersen. Teaching object-oriented programming – towards teaching a systematic programming process. In *Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts. Affiliated with 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, 2004.

[5] J. Bennedsen and C. Schulte. Bluej visual debugger for learning the execution of object-oriented programs? *Trans. Comput. Educ.*, 10(2):8:1–8:22, June 2010.

[6] J. Bergin. Fourteen pedagogical patterns. In *Proceedings of the 5th European Conference on Pattern Languages of Programms (EuroPLoP '2000), Irsee, Germany, July 5-9, 2000*, pages 1–49. UVK - Universitaetsverlag Konstanz, 2000.

[7] M. J. Blesa, A. Duch, J. Gabarró, J. Petit, and M. Serna. Continuous assessment in the evolution of a CS1 course: The pass rate/workload ratio. In S. Zvacek, M. T. Restivo, J. Uhomoibhi, and M. Helfert, editors, *Computer Supported Education*, pages 313–332, Cham, 2016. Springer International Publishing.

[8] N. C. C. Brown, A. Altadmri, S. Sentance, and M. Kölling. Blackbox, five years on: An evaluation of a large-scale programming data collection project. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 196–204, New York, NY, USA, 2018. ACM.

[9] P. Burton. Kinds of language, kinds of learning. *Sigplan Notices*, 33:53–61, 1998.

[10] W. Cazzola and D. M. Olivares. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, 2016.

[11] G. Echeita Sarrionandia and C. Duk Homad. Inclusión educativa. *REICE. Revista Electrónica Iberoamericana sobre Calidad, Eficacia y Cambio en Educación*, 2008.

[12] A. Eckerdal and M. Thuné. Novice java programmers' conceptions of "object" and "class", and variation theory. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 89–93, New York, NY, USA, 2005. ACM.

[13] V. Efopoulos, V. Dagdilelis, G. Evangelidis, and M. Satratzemi. Wipe: A programming environment for novices. *SIGCSE Bull.*, 37(3):113–117, June 2005.

[14] Estefania Miguel, Miguel Carboni, and Nicolás Passerini. Hoope, construyendo un lenguaje para enseñar, Nov. 2013.

[15] S. Georgantaki and S. Retalis. Using educational tools for teaching object oriented design and programming. *Journal of Information Technology Impact*, 7:111–130, 01 2007.

[16] M. N. Giannakos, I. O. Pappas, L. Jaccheri, and D. G. Sampson. Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness. *Education and Information Technologies*, 22(5):2365–2382, 2017.

[17] H. Goldenhersh, A. Coria, and M. Saino. Deserción estudiantil: desafíos de la universidad pública en un horizonte de inclusión. *Revista Argentina de Educación Superior*, (3):97–120, 2011.

[18] K. E. Gray and M. Flatt. Professorj: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177. ACM, 2003.

[19] C. Griggio, G. Leiva, G. Polito, G. Decuzzi, and N. Passerini. A programming environment supporting a prototype-based introduction to OOP. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[20] M. Guzdial. Centralized mindset: a student problem with object-oriented programming. volume 27, pages 182–185, 01 1995.

[21] D. Hagan and S. Markham. Teaching java with the bluej environment. In *Ascilite (Australasian Society for Computers in Learning in Tertiary Education) Conferense*, 2000.

[22] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 131–134, New York, NY, USA, 1997. ACM.

[23] T. Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer, 2002.

[24] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[25] S. Kouznetsova. Using bluej and blackjack to teach object-oriented design concepts in cs1. *J. Comput. Sci. Coll.*, 22(4):49–55, Apr. 2007.

[26] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOOP*, 11(8):8–15, 1999.

[27] C. Lombardi, N. Passerini, and L. Cesario. Instances and classes in the introduction of object oriented programming. In *Smalltalks 2007 - 1st conference on Smalltalk based technologies, research and industry applications*, DC - FCEN - UBA, Buenos Aires, Argentina, Dec. 2007.

[28] B. Meyer. The outside-in method of teaching introductory programming. In M. Broy and A. Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 66–78. Springer Berlin Heidelberg, 2003.

[29] N. Passerini, C. Lombardi, J. Fernandes, P. Tesone, and F. Dodino. Wollok: Language + ide for a gentle and industry-aware introduction to oop. In *2017 Twelfth Latin American Conference on Learning Technologies (LACLO)*, pages 1–4, Oct 2017.

[30] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3):203–221, 2005.

[31] A. W. Schmolitzky and T. Göttel. Guess my object: An 'objects first' game on objects' behavior and implementation with bluej. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 219–224, New York, NY, USA, 2014. ACM.

[32] J. E. Sánchez-García, M. Urías-Ruiz, and B. E. Gutiérrez-Herrera. Análisis de los problemas de aprendizaje de la programación orientada a objetos. *Ra Ximhai*, 11(4):148–175, 2015.

[33] M. P. Uysal. The effects of objects-first and objects-late methods on achievements of OOP learners. *Journal of Software Engineering and Applications*, 5(10):816–822, 2012.

[34] L. S. Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

[35] S. Xinogalos. Object-oriented design and programming: An investigation of novices' conceptions on objects and classes. *TOCE*, 15(3):13:1–13:21, 2015.

[36] S. Xinogalos, M. Satratzemi, and V. Dagdilelis. Re-designing an OOP course based on bluej. In *ICALT*, pages 660–664. IEEE Computer Society, 2007.